PROJET DE SUPPORT DE COURS-TD POUR LE MODULE OPTIONNEL "INFORMATIQUE ET MATHÉMATIQUES"

J. Sauloy [1]

13 juin 2011

1. U.F.R. M.I.G., Université Paul Sabatier, 118, route de Narbonne, 31062 Toulouse CEDEX 4

Table des matières

Conventions et notations générales

Conventions. La notation $A := B$ signifiera que le terme A est défini par la formule B. Les expressions nouvelles sont écrites en *italiques* au moment de la définition. Noter qu'une définition peut apparaitre au cours d'un théorème, d'un exemple, d'un exercice, etc.

Exemple 0.0.1 L'entier $r := a - qb$ est appelé *reste de la division euclidienne de a par b*.

Notations. Voici les plus courantes (parmi celles qui ne sont pas totalement standardisées et ne sont pas introduites dans le cours) :

- Si $x \in \mathbf{R}$, l'unique entier n tel que $n \leq x < n+1$ est noté $\lfloor x \rfloor$ et appelé *partie entière de x*.
- si $a \in \mathbf{R}_+$ et $b \in \mathbf{N}$, $b \geq 2$, on note $\log_b(a) := \dfrac{\ln a}{\ln b}$ le logarithme en base b de a.
-
-
-

Chapitre 1

Arithmétique élémentaire

1.1 Prélude : la division euclidienne dans Z

Théorème 1.1.1 *Soient $a \in \mathbf{Z}$ et $b \in \mathbf{N}^*$. Il existe alors un unique couple $(q,r) \in \mathbf{Z} \times \mathbf{Z}$ tel que :*

$$\begin{cases} a = qb + r, \\ 0 \leq r \leq b-1. \end{cases} \tag{1.1.1.1}$$

Les entiers q et r sont respectivement appelés quotient *et* reste *de la* division euclidienne de a par *b. (Et l'on dit parfois que a est le dividende et b le diviseur.)*

Preuve. - On commence par l'unicité. Si $a = qb + r = q'b + r'$ avec $0 \leq r, r' \leq b-1$, on a d'une part $-(b-1) \leq r' - r \leq (b-1)$, autrement dit, $|r' - r| \leq (b-1)$; d'autre part $r' - r = (q - q')b$. Cela entraine $r' - r = 0$ (seul multiple de b de valeur absolue strictement inférieure à b) donc $q - q' = 0$. Pour prouver l'existence, on va utiliser un algorithme :

```
q := 0;
r := a;
Si (a >= 0) alors
   tant que (r >= b) faire
      r := r - b;
      q := q + 1;
   fin tant que;
sinon
   tant que (r < 0) faire
      r := r + b;
      q := q - 1;
   fin tant que;
fin si;
rendre (q,r);
```

Tout d'abord, la terminaison de l'algorithme est garantie : dans le cas où $a \geq 0$, le “compteur” r décroit strictement à chaque étape, donc la condition de continuation $r \geq b$ finit par être fausse ; et dans le cas contraire, le “compteur” r croit strictement à chaque étape, donc la condition de continuation $r < 0$ finit par être fausse.

Pour prouver la correction de l'algorithme, supposons d'abord que l'on est dans le premier cas $a \geq 0$. On introduit l'invariant de boucle :

$$a = qb + r \text{ et } r \geq 0.$$

Il est vérifié à l'initialisation. De plus, chaque exécution de l'instruction `(r := r - b; q := q + 1)` sous la condition de continuation `r >= b` le conserve. En effet, notant r', r'' les états avant et après de r et q', q'' les états avant et après de q, on a : $r'' = r' - b$ et $q'' = q' + 1$, donc $q''b + r'' = (q'+1)b + (r'-b) = q'b + r'$, d'où la conservation de la condition $a = qb + r$. D'autre part, $r' \geq b$ (condition d'exécution de la boucle) donc $r'' \geq 0$.
A la sortie de la boucle `tant que`, on a $r < b$ (la condition de sortie est la négation de la condition `r >= b`) et *en plus* l'invariant de boucle $a = qb + r$ et $r \geq 0$. C'est exactement équivalent aux conditions énoncées par le théorème : $a = qb + r$ et $0 \leq r \leq b - 1$.
Supposons maintenant que l'on est dans le deuxième cas $a \geq 0$. On introduit l'invariant de boucle :

$$a = qb + r \text{ et } r < b.$$

Le raisonnement est alors similaire, on laisse au lecteur le plaisir de le rédiger. □

On notera par la suite `divEucl(a,b)` le résultat de la division euclidienne de a par b, donc les deux nombres q et r calculés par l'algorithme ci-dessus.

Corollaire 1.1.2 *Le quotient de la division euclidienne de a par b est $q = \lfloor a/b \rfloor$.*

Preuve. - Puisque r et b sont des entiers, la condition $r \leq b - 1$ est équivalente à la condition $r < b$. L'entier q est alors défini par la condition :

$$0 \leq a - qb \leq b - 1 \iff 0 \leq a - qb < b \iff qb \leq a < (q+1)b \iff q \leq a/b < (q+1),$$

d'où la conclusion. □

Bien qu'un algorithme ait servi à établir l'existence du quotient et du reste, nous ne prétendons pas que les systèmes de calcul existants (comme par exemple Maple) utilisent réellement cet algorithme pour effectuer une division euclidienne. Dans la suite de ce chapitre, nous éluderons cette question et admettrons l'existence d'une "fonction primitive" qui prend en argument le couple (a,b) et donne comme résultat le couple (q,r) ; mode d'emploi : $(q,r) := \text{diveucl}(a,b)$. *Nous admettrons de plus que ce calcul s'effectue en temps constant.*

Exercice 1.1.3 On appelle *mantisse* de $x \in \mathbf{R}$ le réel $M(x) := x - \lfloor x \rfloor$.
(i) Montrer que c'est l'unique réel $\alpha \in [0,1[$ tel que $x - \alpha \in \mathbf{Z}$.
(ii) Exprimer le reste de la division euclidienne de a par b en fonction de $M(a/b)$.

TP 1.1.4 Programmer la fonction `diveucl` en utilisant l'algorithme décrit dans la preuve du théorème.

1.2 Calcul en base b

Dans toute cette section, la *base* b est un entier naturel fixé tel que $b \geq 2$. (Le cas $b = 1$ correspond à l'écriture en bâtons : 1 mammouth = 1 bâton.) Nous ne traiterons que le cas des entiers naturels : donc, ni les entiers négatifs, ni les nombres non entiers.

1.2.1 Ecriture en base b

Rappelons le principe de l'écriture décimale : le nombre noté 2011 est égal à $2.10^3 + 0.10^2 + 1.10^1 + 1.10^0$. Le même nombre admet l'écriture binaire 11111011011, ce qui signifie $1.2^{10} + 1.2^9 + 1.2^8 + 1.2^7 + 1.2^6 + 0.2^5 + 1.2^4 + 1.2^3 + 0.2^2 + 1.2^1 + 1.2^0$. Si l'on ne veut pas confondre cette écriture binaire 11111011011 avec celle d'un entier de l'ordre de dix milliards, on écrira plutôt $(11111011011)_2$. Notons que, dans les deux cas, nous écrivons les puissances (de 10 ou de 2) par ordre décroissant d'exposant : le "chiffre des unités" est à droite.

Avant d'énoncer et de démontrer le théorème fondamental, voici un résultat préliminaire qui sert beaucoup dans la pratique.

Lemme 1.2.1 *Soient $k \geq 0$ un entier et $c_0, \dots, c_k \in \{0, \dots, b-1\}$. On suppose que $c_k \neq 0$. L'entier $a := c_k b^k + \cdots + c_0 b^0$ vérifie alors :*

(1.2.1.1) $$b^k \leq a \leq b^{k+1} - 1.$$

Preuve. - Puisque $c_i \geq 0$ et $c_k \geq 1$, un minorant de a est :

$$1.b^k + 0.b^{k-1} + \cdots + 0.b^0 = b^k.$$

Puisque $c_i \leq b-1$, un majorant de a est :

$$(b-1)b^k + \cdots + (b-1)b^0 = b^{k+1} - 1.$$

□

Théorème 1.2.2 *Soit $a \in \mathbf{N}^*$ un entier naturel non nul. Il existe alors un unique entier $k \geq 0$ et une unique suite finie $(c_0, \dots, c_k)$ tels que :*

(1.2.2.1) $$\begin{cases} a = c_k b^k + \cdots + c_0 b^0, \\ 0 \leq c_i \leq b-1 \text{ pour } i = 0, \dots, k, \\ c_k \neq 0. \end{cases}$$

On dit alors que $(c_k \cdots c_0)_b$ est l'écriture de a en base b. *Les c_i sont les* chiffres *de cet écriture. On précise parfois que cette écriture se fait* poids forts à gauche.

Preuve. - L'unicité se prouve par "récurrence forte" sur a. Si $a \leq b-1$, d'après le lemme, on a nécessairement $k = 0$, d'où l'on déduit $c_0 = a$. L'unicité est donc bien démontrée dans ce cas.
Soit donc $a \geq b$ et supposons l'unicité prouvée pour tout entier a' tel que $a' < a$. (C'est à cause de cette forme particulière de l'hypothèse de récurrence que l'on parle de "récurrence forte".) Si l'on a une écriture $a = c_k b^k + \cdots + c_0 b^0$, on en déduit :

$$a = a'b + c_0 \text{ avec } a' = c_k b^{k-1} + \cdots + c_1 b^0.$$

On voit donc que c_0 est *nécessairement* le reste de la division euclidienne de a par b ; que a' est *nécessairement* le quotient de la division euclidienne de a par b ; et que $(c_k \cdots c_1)_b$ est *nécessairement* une écriture de a' en base b. Mais, d'après le lemme, $a' \leq b^k - 1 < a$. D'après l'hypothèse

de récurrence, l'écriture de a' en base b est unique, il en est donc de même de l'écriture de a.
L'analyse ci-dessus nous suggère l'algorithme dont nous déduirons l'existence de l'écriture en base b : il consiste en des divisions euclidiennes successives. Le résultat est retourné sous la forme d'un tableau c tel que $a = \sum_{i=0}^{k} c[i]b^i$:

```
x := a;
i := 0;
tant que (x >= b) faire
   (q,r) := divEucl(x,b);
   c[i] := r;
   x := q;
   i := i + 1;
fin tant que;
c[i] := x;
k := i;
rendre (k,c);
```

La terminaison est garantie par le fait que x décroit strictement, donc que la condition de continuation `x >= b` finit par être fausse.
La correction repose sur l'invariant de boucle :

$$a = c[0]b^0 + \cdots + c[i-1]b^{i-1} + xb^i.$$

Nous laissons au lecteur le soin de prouver la validité de cet invariant et d'en déduire la correction de l'algorithme. □

Plusieurs remarques s'imposent ici :

1. Nous éludons totalement la question de la représentation "interne" de a, de b et des chiffres c_i ; et donc également de la manière dont sont implémentés les calculs de division euclidienne.
2. Nous avons mis à part le cas de 0 qui, quelle que soit la base, s'écrit 0.
3. Nous avons défini et calculé l'écriture *propre* de a. On aura parfois besoin de ses écritures *impropres*, dans lesquelles on s'autorise à ajouter des chiffres 0 au delà du chiffre de poids fort : par exemple $(00011111011011)_2$ au lieu de $(11111011011)_2$. Lorsque l'on autorise les écritures impropres, il n'y a plus unicité de l'écriture.

Exercice 1.2.3 On propose un traitement mathématique de l'invariant de boucle. On pose $x_0 := a$ et, tant que $x_i > 0$, $(x_{i+1}, c_i) := \text{diveucl}(x_i, b)$. Démontrer par récurrence que, pour tout i :

$$a = c_0 b^0 + \cdots + c_{i-1} b^{i-1} + x_i b^i.$$

TP 1.2.4 Programmer le calcul de l'écriture en base b d'un entier a. A défaut de savoir représenter les tableaux, on écrira un programme qui affiche les chiffres de a puis l'entier k.

Taille du codage d'un entier en base b

L'écriture en base b sert avant tout à *coder* les entiers. Le codage d'un entier donné représente un certain espace dans la mémoire d'un ordinateur. Nous *estimerons* cet espace mémoire en supposant (pour simplifier notre modèle mathématique) qu'il est entièrement occupé par les chiffres. Cette hypothèse simplificatrice est d'ailleurs une approximation de la réalité puisqu'il faut bien indiquer quelque part que l'on est au bout de l'écriture de cet entier.

Remarque 1.2.5 Une autre possibilité serait que tous les entiers soient codés dans des espaces de même taille, donc en écriture impropre avec le même nombre total de chiffres. C'est ce qui se pratique quand les entiers manipulés ne sont pas trop grands. Dans cette section, nous supposerons que les entiers peuvent être très grands et que la taille de leur codage doit être estimée correctement.

Nous supposerons également que tous les chiffres de l'écriture en base b d'un entier occupent des espaces de mêmes tailles : notons K l'espace occupé par un chiffre (disons que c'est un nombre de bits). Si $a = c_0 + \cdots + c_k b^k$, avec $c_k \neq 0$, l'espace total occupé par a est : $(k+1)K$.

Proposition 1.2.6 *Le nombre de chiffres de l'écriture de a en base b est égal à* $1 + \lfloor \log_b a \rfloor$.

Preuve. - Ce nombre de chiffres est $k+1$; or, d'après le lemme 1.2.1 :

$$b^k \leq a \leq b^{k+1} - 1 \Longrightarrow b^k \leq a < b^{k+1} \Longrightarrow k \leq \frac{\ln a}{\ln b} < k+1 \Longrightarrow k = \lfloor \frac{\ln a}{\ln b} \rfloor.$$

□

Pour ne pas avoir à tenir compte de l'unité de taille mémoire (ici les K bits occupés par chaque chiffre), nous ne prendrons en compte dans l'estimation de la taille de a que le nombre de chiffres :

Définition 1.2.7 La *taille* d'un entier $a \in \mathbf{N}^*$ (sous-entendu : écrit en base b) est le nombre de ses chiffres, $1 + \lfloor \log_b a \rfloor$. On la notera $|a|$ (donc sans préciser la base b relativement à laquelle cette taille est calculée).

Exercice 1.2.8 Si l'on écrit en base 2, un chiffre est un bit et $K = 1$. Si l'on écrit en base 16 (système *hexadécimal*), on peut supposer qu'un chiffre s'écrit sur quatre bits, et $K = 4$. Comparer les espaces mémoire occupés par un même entier dans ces deux représentations.

Le "coût" de l'algorithme de codage en base b

Ici et dans toute la suite de ce cours, nous appellerons *coût* d'un algorithme le temps d'exécution d'un programme implémentant cet algorithme. Trois points très importants sont à prendre en compte :

1. Ce temps dépend presque toujours de la *taille* des données, et nous le considèrerons donc comme une fonction de cette taille. Par exemple : le temps total de multiplication de deux entiers par la méthode apprise à l'école primaire dépend du nombre de chiffres de ces entiers.
2. Nous ne mesurons pas ce temps mais tentons de le prédire à partir d'une analyse du comportement dynamique de l'algorithme. Par exemple : pour calculer a^n, un algorithme simplet effectuera $(n-1)$ multiplications.

3. Selon la machine utilisée, le langage de programmation, voire l'environnement d'exécution, les opérations élementaires (opérations arithmétiques, mais aussi : comparaisons entre nombres, lectures et écritures, affectations, etc) ne prennent pas le même temps. Nous n'en tiendrons pas compte et remplacerons ces temps élémentaires par des constantes.

Comme déjà indiqué, nous supposons que le coût d'une division euclidienne est constant. Il est raisonnable de supposer que toutes les autres instructions écrites dans l'algorithme s'effectuent également en temps constant (il y a des additions, des affectations, etc). Le temps total d'exécution est donc de la forme $Ak+B$, où k est l'entier qui apparait dans l'écriture de a ; en effet, le nombre d'exécutions de la boucle interne (c'est-à-dire le nombre de fois que l'on aura $x \geq b$) est égal au nombre de fois où l'on calcule un chiffre c_i en ne comptant pas le dernier chiffre c_k (qui est calculé à la fin).

Le *coût* de l'algorithme de codage en base b est donc une fonction affine [1] de la taille de l'entier codé : en effet, avec les mêmes notations, $|a| = k+1$, d'où :

$$Ak+B = A\,|a| + B', \text{ où } B' = B - A.$$

Pour des entiers de grande taille k, c'est presque la même chose que si l'on avait un coût de la forme $A\,|a|$, et il est d'usage de dire plus simplement que *le coût (du codage) est une fonction linéaire de la taille*.

1.2.2 Addition en base b

Soient $a = c_0 + \cdots + c_k b^k$ et $a' = c'_0 + \cdots + c'_{k'} b^{k'}$ les écritures propres des deux nombres à additionner. Supposons par exemple que $k' < k$. On peut alors prolonger l'écriture de a' en une écriture impropre en posant $a'_{k'+1} := 0, \ldots, a'_k := 0$, d'où $a' = c'_0 + \cdots + c'_k b^k$. Cela permet dans tous les cas d'additionner deux nombres écrits avec $(k+1)$ chiffres, où $k+1 = \max(|a|, |a'|)$.

En première approche, on a :

$$a + a' = (c_0 + c'_0) + \cdots + (c_k + c'_k) b^k.$$

C'est facile à écrire, malheureusement ce n'est en général pas une écriture en base b puisque l'on peut avoir $c_i + c'_i > b$. (Il y a même un peu plus d'une chance sur deux que cela se produise pour un i donné, donc, si k est grand, il est extrêmement probable que cela se produise pour un i au moins !)

Dans tous les cas, $c_i + c'_i \leq 2b-2$. Supposons par exemple que $b \leq c_0 + c'_0 \leq 2b-2$. On écrira alors :

$$(c_0+c'_0)+(c_1+c'_1)b = (c_0+c'_0-b)+(c_1+c'_1+1)b \Longrightarrow a+a' = (c_0+c'_0-b)+(c_1+c'_1+1)b+\cdots+(c_k+c'_k)b^k.$$

Le problème se repose alors avec $c_1 + c'_1 + 1$: ce nombre est *a priori* compris entre 0 et $2b-1$, mais s'il est supérieur ou égal à b on doit recommencer la même manoeuvre. Le lecteur aura reconnu le problème de la *retenue*. Pour finir, on obtient l'algorithme appris à l'école primaire d'addition en base b de deux nombres $\sum_{i=0}^{k} c[i] b^i$ et $\sum_{i=0}^{k} c'[i] b^i$ pour produire le résultat $\sum_{i=0}^{l} d[i] b^i$:

1. Fonction linéaire : $f(x) = \alpha x$; fonction affine : $f(x) = \alpha x + \beta$.

```
i := 0;
r := 0;
tant que (i <= k)
   d[i] := c[i] + c′[i] + r;
   si (d[i] >= b) alors
      d[i] := d[i] - b;
      r := 1;
   sinon
      r := 0;
   fin si;
   i := i + 1;
fin tant que;
si (r = 0) alors
   l := k;
sinon
   l := k+1;
   d[l] := r;
fin si;
rendre (d);
```

L'entier l apparu à la fin est le nombre de chiffres de la somme : donc k ou $k+1$, selon qu'il y avait ou non une retenue à la toute dernière addition.

Il est évident que cet algorithme se termine après $(k+1)$ exécutions de l'instruction de boucle. La correction se prouve à l'aide de l'invariant de boucle :

$$(c_0 + \cdots + c_i b^i) + (c'_0 + \cdots + c'_i b^i) = (d_0 + \cdots + d_i b^i) + r b^{i+1}.$$

On doit ajouter le fait que, à tout moment, $r = 0$ ou 1, donc que $c_i + c'_i + r \leq 2b - 1$; donc le d_i calculé est bien un chiffre (un nombre entre 0 et $b-1$).

Exercice 1.2.9 Quelle est la probabilité qu'il n'y ait aucune retenue ?

TP 1.2.10 Programmer l'addition en base b.

Coût de l'algorithme d'addition en base b

Chaque étape de la boucle "tant que" contient une addition de deux chiffres, une addition de la retenue, un test, peut-être une soustraction, certainement une incrémentation. En prenant en compte les instructions avant et après la boucle "tant que", on conclut qu'il existe des constantes $A, A' > 0$ et des constantes B, B' telles que le coût soit compris entre $A(k+1) + B$ et $A'(k+1) + B'$, autrement dit, entre $A \max(|a|, |a'|) + B$ et $A' \max(|a|, |a'|) + B'$.

De manière très informelle, on peut encore dire que le coût (de l'addition) est une fonction "à peu près" linéaire de la taille des données. On peut même préciser cet à-peu-près comme suit :

1. L'ordre de grandeur du coût n'est pas pire qu'un coût linéaire. Plus précisément, lorsque la taille des données a, a' est de plus en plus grande, le rapport :

$$\frac{\text{coût de l'addition } a+a'}{\max(|a|,|a'|)}$$

ne croît pas indéfiniment, il reste majoré. L'écriture mathématique de cette propriété est :

$$\text{coût de l'addition } a+a' = O\big(\max(|a|,|a'|)\big).$$

2. L'ordre de grandeur du coût n'est pas non plus meilleur qu'un coût linéaire. Plus précisément, lorsque la taille des données a, a' est de plus en plus grande, le rapport :

$$\frac{\max(|a|,|a'|)}{\text{coût de l'addition } a+a'}$$

ne croît pas indéfiniment, il reste majoré. Avec la convention précédente, on a donc :

$$\max(|a|,|a'|) = O\big(\text{coût de l'addition } a+a'\big).$$

3. Pour dire que l'ordre de grandeur du coût n'est ni pire ni meilleur qu'un coût linéaire, on résume les deux propriétés précédentes par la formule :

$$\text{coût de l'addition } a+a' = \Theta\big(\max(|a|,|a'|)\big).$$

Exercice 1.2.11 L'entier $k := \max(|a|,|a'|)$ étant fixé, décrire un cas où l'addition se fait à coût minimum, resp. à coût maximum, et calculer précisément ce minimum et ce maximum.

1.2.3 Multiplication en base b

Multiplication d'un nombre par un chiffre

Soient $a = c_0 + \cdots + c_k b^k$ (écriture propre en base b) et $c \in \{0,\ldots,b-1\}$. Dans le cas où $c = 0$ ou 1, le produit ca se calcule sans trop de peine ... En général, on calcule les produits cc_i et l'on doit s'occuper des retenues.

Nous admettrons que le produit de deux chiffres (tables de multiplication de l'école primaire ! ! !) se fait en temps constant : il peut être tabulé (les résultats sont en mémoire) ou cablé (réalisé par du hardware) ou réalisé par un tout petit programme *ad hoc*.

Le résultat de cette multiplication est inférieur ou égal à $(b-1)^2$: nombre à deux chiffres, qui s'écrit $qb+r$ avec $q \leq b-2$ (puisque $(b-1)^2 < (b-1)b$). *Attention !* c'est q qui va servir de retenue et r sera le chiffre à écrire. En effet, par exemple :

$$cc_0 = qb + r \Longrightarrow c(c_0 + c_1 b) = d_0 + d_1 b \text{ avec } d_0 = r \text{ et } d_1 = cc_1 + q.$$

Naturellement, on doit encore effectuer une division euclidienne de $cc_1 + q$ par b pour calculer le chiffre d_1 et la prochaine retenue ; mais $cc_1 + q \leq (b-1)^2 + b - 2 < b^2 - b$, et le quotient (prochaine retenue) sera encore inférieur ou égal à $b-2$. Voici l'algorithme (nous notons maintenant s la retenue). L'algorithme ci-dessous de multiplication du nombre $\sum_{i=0}^{k} c[i]b^i$ par le chiffre c a pour résultat $\sum_{i=0}^{l} d[i]b^i$:

```
i := 0;
s := 0;
tant que (i <= k)
   (q,r) := divEucl(c*c[i] + s,b);
   d[i] := r;
   s := q;
   i := i+1;
fin tant que;
si s = 0
  alors l := k
  sinon (l := k + 1; d[l] := s);
fin si;;
```

Le coût de cet algorithme est linéaire en $|a|$.

Multiplication de deux nombres quelconques

Pour multiplier les entiers $a = c_0 + \cdots + c_k b^k$ et $a' = c'_0 + \cdots + c'_{k'} b^{k'}$, l'algorithme appris en primaire (multiplications d'un nombre par un chiffre avec décalages vers la gauche et additions) revient à calculer successivement : ac'_0, puis $ac'_1 b$, puis $ac'_2 b^2$, etc ; et à additionner le tout : dans ce cas, il s'agit donc d'additionner les $(k'+1)$ produits $ac'_i b^i$. Bien entendu, à cette heureuse époque (l'école primaire), l'addition se faisait directement. Pour écrire un programme, il est cependant plus simple de faire des additions successives de deux nombres. Voici donc le *schéma* de l'algorithme de multiplication en base b de deux nombres $\sum_{i=0}^{k'} c'[i] b^i$ et a pour produire le résultat s :

```
s := 0;
i := 0;
tant que (i <= k') faire
   s := s + c'[i]*a*b^i;;
fin tant que;
rendre (s);
```

Naturellement, la multiplication `c'[i]*a` se fait selon l'algorithme vu plus haut ; et la multiplication du résultat par b^i n'en est pas réellement une, il s'agit simplement d'un *décalage* (avec ajout de chiffres 0 à droite).

On peut démontrer que le coût de la multiplication est linéaire en le produit des tailles :

$$\text{coût de la multiplication } a.a' = \Theta\big(|a| \times |a'|\big).$$

Exercice 1.2.12 Le prouver.

TP 1.2.13 Programmer la multiplication et l'expérimenter pour vérifier l'estimation annoncée du coût.

1.3 Exponentiation rapide

Il s'agit de calculer les puissances a^n, $n \in \mathbf{N}^*$, d'un nombre a qui peut être compliqué : réel en écriture flottante, grand entier ... Un réflexe fréquent pour calculer a^n avec une calculette en évitant toute programmation est d'écrire (en supposant bien entendu que $a > 0$) :

$$a^n = \exp(n \ln a).$$

Cette méthode est aberrante. En effet, le calcul des fonctions dites "transcendantes" comme l'exponentielle et le logarithme (ainsi que les fonctions trigonométriques) fait appel à des sous-programmes qui effectuent un *grand nombre* de calculs et qui, dans tous les cas, rendent un résultat *approché*. Le lecteur est vivement encouragé à calculer 3^3 par cette méthode et à tenter d'estimer le temps de calcul et la précision du résultat.

TP 1.3.1 Faire l'expérience : calculer un million de fois $3 \times 3 \times 3$ et $\exp(3 \ln 3)$.

1.3.1 Méthode naturelle

Il s'agit bien entendu de poser (on suppose ici $n \geq 2$) :

$$a^n = a \times \cdots \times a,$$

où il y a n facteurs et donc $(n-1)$ symboles d'opérations $\times$. Le calcul représente donc $(n-1)$ multiplications. En admettant que le coût de ces multiplications est constant, on en déduit :

$$\text{coût du calcul de } a^n = \Theta(n).$$

Naturellement, l'unité de coût, c'est-à-dire le coût unitaire de chaque multiplication, dépend ici d'autres facteurs :

- Type du nombre a : la multiplication de réels en écriture flottante est plus chère que la multiplication d'entiers courts.
- Algorithmes utilisés par les librairies numériques du langage, du système d'exploitation.
- Vitesse du hardware.

Exercice 1.3.2 On admet que le coût de la multiplication est linéaire en le produit des tailles :

$$\text{coût de la multiplication } a.a' = \Theta\big(|a| \times |a'|\big).$$

Montrer que le coût de calcul de a^n est alors $\Theta(n^2)$.

1.3.2 Méthode dichotomique

Cette méthode très ancienne est appelée exponentiation *dichotomique* ou *indienne* ou *chinoise* ou *babylonienne* ... On la comprend mieux sur un exemple. Le calcul de a^8, qui devrait coûter 7 multiplications, peut se réaliser en 3 multiplications comme suit :

$$x_1 := a \times a, \quad x_2 := x_1 \times x_1, \quad x_3 := x_2 \times x_2.$$

Remarque 1.3.3 Il y a un (petit) prix à payer pour ce gain de performance : l'utilisation de variables pour désigner les calculs intermédiaires cache en réalité l'utilisation d'un peu de mémoire pour stocker ces résultats.

Naturellement, l'exposant 8 étant pair, la puissance $a^8 = (a^4)^2$ est un carré. La situation n'est pas toujours aussi favorable. Par exemple, le calcul de a^7, qui aurait coûté 6 multiplications par la méthode naturelle, peut se réaliser en 4 multiplications, donc plus que pour a^8 ! La séquence est la suivante :

$$x_1 := a \times a, \quad x_2 := x_1 \times a, \quad x_3 := x_2 \times x_2, \quad x_4 := x_3 \times a.$$

Il a fallu compléter le carré $x_1 = a^2$ en le multipliant par a (à cause de l'exposant impair 3) ; puis compléter le carré $x_3 = a^6$ en le multipliant par a (à cause de l'exposant impair 7). Cela explique le relatif surcoût. On s'attend en général à réaliser des gains de performance en accord avec le slogan : "plus il y a d'exposants pairs, plus on y gagne". Nous allons tout de même vérifier que la méthode est efficace dans tous les cas, autrement dit que l'on a une "performance asymptotique" infiniment meilleure que par la méthode naturelle (ces termes seront expliqués).

Tout d'abord, explicitons informellement la méthode. Un algorithme précis sera écrit plus loin.

- Si $n = 2p$, calculer $b := a^p$ puis $a^n := b \times b$.
- Si $n = 2p+1$, calculer $b := a^p$ puis $a^n := b \times b \times a$.

Notons $c(n)$ le nombre de multiplications nécessaires pour calculer a^n. Par exemple $c(1) = 0$ et, de toute évidence, $c(2) = 1$. De la description ci-dessus, on déduit les règles :

$$\begin{aligned} c(2p) &= c(p) + 1, \\ c(2p+1) &= c(p) + 2. \end{aligned}$$

Cela permet déjà de calculer les premières valeurs de $c(n)$:

n	1	2	3	4	5	6	7	8	9	10	11	12
$c(n)$	0	1	2	2	3	3	4	3	4	4	5	4
n	13	14	15	16	17	18	19	20	21	22	23	24
$c(n)$	5	5	6	4	5	5	6	5	6	6	7	5

Nous allons maintenant trouver la formule générale donnant le coût $c(n)$. Pour cela, on prend en compte *l'écriture binaire* des expressions $2p$ et $2p+1$. En effet, si l'écriture binaire de p est $(b_k \cdots b_0)_2$, alors :

- L'écriture binaire de $2p$ est $(b_k \cdots b_0 0)_2$;
- L'écriture binaire de $2p+1$ est $(b_k \cdots b_0 1)_2$.

Comme $c(1) = 0$, on en tire la règle suivante :

Proposition 1.3.4 *Soit $(a_k \cdots a_0)_2$ l'écriture binaire propre de n (donc $a_k = 1$). Soient α le nombre de 0 et β le nombre de 1 parmi les bits $a_0, \ldots, a_{k-1}$ (on ne prend donc pas en compte le bit de poids fort a_k). Alors :*

$$c(n) = \alpha + 2\beta = k + \beta.$$

Preuve. - En effet, on peut reconstituer l'écriture binaire propre de n en partant du bit 1 (écriture binaire du nombre 1), correspondant au coût $c(1) = 0$, et en rajoutant successivement à droite des bits 0 et des bits 1.

– Chaque fois que l'on rajoute un bit 0, le coût augmente de 1 ; cela se produit α fois.
– Chaque fois que l'on rajoute un bit 1, le coût augmente de 2 ; cela se produit β fois.

On a donc bien à la fin $c(n) = \alpha + 2\beta$. D'autre part, le nombre total de bits que l'on a ajouté est $\alpha + \beta = k$, d'où la deuxième formule. □

Corollaire 1.3.5 *(i) Le cas le meilleur est celui où l'on n'a rajouté que des* 0 *: alors* $n = 2^k$ *et* $c(n) = k$.
(ii) Le cas le pire est celui où l'on n'a rajouté que des 1 *: alors* $n = 2^{k+1} - 1$ *et* $c(n) = 2k$.
(iii) Dans tous les cas, on a $k \leq c(n) \leq 2k$.

Rappelons d'autre part que $k = \lfloor \log_2 n \rfloor = |n|_2 - 1$, où l'on désigne par $|n|_2$ la taille de l'écriture de n en base 2.

Corollaire 1.3.6 *(i) Le coût* $c(n)$ *vérifie :*

$$\lfloor \log_2 n \rfloor \leq c(n) \leq 2\lfloor \log_2 n \rfloor.$$

(ii) On a $c(n) = \Theta(\log_2 n) = \Theta(|n|_2)$.

Comme $\lim_{n \to +\infty} \frac{\log_2 n}{n} = 0$, on voit que, pour de grandes valeurs de l'exposant n, *le rapport du coût par la méthode dichotomique au coût par la méthode naturelle tend vers* 0 : c'est le sens de la phrase "on a une performance asymptotique infiniment meilleure que par la méthode naturelle".

L'algorithme

On traduit la méthode décrite informellement plus haut par l'algorithme de calcul de $r := x^n$:

```
r := 1;
x := a;
p := n;
tant que (p > 0)
   si p impair alors
      r := r*x;
   fin si;
   x := x * x;
   p := p div 2;
fin tant que;
rendre (r);;
```

(On a noté par `p div 2` le quotient de la division de p par 2.)

Si $p > 0$, le quotient de la division euclidienne par 2 vérifie $p \div 2 < p$. Dans l'algorithme, l'entier p décroit donc strictement à chaque étape, ce qui garantit la terminaison.

L'invariant de boucle qui permet de prouver la correction est le suivant : après chaque étape, on a $a^n = rx^p$ et $p \geq 0$. Nous laissons au lecteur de vérifier que ces conditions sont conservées. En fin d'itération, on n'a plus $p > 0$, donc $p = 0$ et $r = a^n$.

Exercice 1.3.7 L'algorithme écrit ci-dessus effectue une multiplication inutile à la fin. Rectifier ce petit défaut (qui affecte la performance mais pas la correction).

TP 1.3.8 Programmer cet algorithme.

1.3.3 Questions d'optimalité

On peut faire un peu mieux au cas par cas

Pour calculer a^{15} par la méthode dichotomique, il faut $c(15) = 6$ multiplications. Voici deux séquences de calcul qui donnent le résultat en 5 multiplications. La première revient à écrire $a^{15} = (a^3)^5$:

$$x_1 := a \times a, \quad x_2 := x_1 \times a, \quad x_3 := x_2 \times x_2, \quad x_4 := x_3 \times x_3, \quad x_5 := x_4 \times x_2.$$

La seconde revient à écrire $a^{15} = (a^5)^3$.

$$x_1 := a \times a, \quad x_2 := x_1 \times x_1, \quad x_3 := x_2 \times a, \quad x_4 := x_3 \times x_3, \quad x_5 := x_4 \times x_3.$$

Ces calculs reposent directement sur une factorisation de n. Il existe une méthode générale pour trouver de telles séquences optimales de calcul, mais elle est compliquée à mettre en oeuvre et le gain n'est pas énorme (voir le paragraphe suivant pour en être sûr). Disons que l'effort en ce sens peut être considéré comme rentable si l'on veut calculer souvent a^{15} ...

On ne peut pas faire beaucoup mieux en général

Nous allons démontrer que l'on ne peut pas espérer calculer a^n en moins de $\log_2 n$ multiplications. La méthode de démonstration consiste à prendre le problème à l'envers et à chercher la plus grande puissance de a que l'on peut calculer en k multiplications.

On fixe les règles comme suit. On pose $x_0 := a$ (qui n'a coûté aucune multiplication). On va calculer $x_1, \dots, x_k$ en utilisant chaque fois une seule multiplication. De plus, le calcul de chaque x_i ne peut faire intervenir que les valeurs déjà calculées : $x_0, \dots, x_{i-1}$. A chaque étape, on calcule donc une puissance $x_i = a^{p_i}$. Bien entendu, $p_0 = 1$ (car $x_0 = a = a^1$) et $p_1 = 2$ (car $x_1 = a \times a = a^2$). Ensuite, tout ce que l'on peut dire, c'est que $x_i = x_j \times x_l$, donc $p_i = p_j + p_l$, avec $0 \leq j, l \leq i - 1$.

Proposition 1.3.9 *(i) On a $p_i \leq 2^i$ pour tout $i = 0, \dots, k$.*
(ii) En particulier, si l'on peut calculer a^n en k multiplications, alors $n \leq 2^k$.

Preuve. - (i) Se prouve par récurrence forte. C'est clair pour $i = 0, 1$. Pour un $i \geq 2$ fixé, supposons que c'est vrai pour tout indice $j \leq i - 1$. On a vu que $p_i = p_j + p_l$, avec $0 \leq j, l \leq i - 1$. Par hypothèse de récurrence (forte), on a $p_j \leq 2^j$ et $p_l \leq 2^l$, d'où :

$$p_i = p_j + p_l \leq 2^j + 2^l \leq 2^{i-1} + 2^{i-1} = 2^i.$$

(ii) en découle en prenant $i = k$ et $n = p_k$. $\square$

Et maintenant, pour le résultat qui nous intéresse, on *renverse* la conclusion :

Corollaire 1.3.10 *Le nombre de multiplications nécessaires pour calculer a^n est supérieur ou égal à $\log_2 n$.*

Preuve. - D'après la proposition, ce nombre k vérifie $n \leq 2^k$, donc $k \geq \log_2 n$. $\square$

Exercice 1.3.11 Pour quelles valeurs de $n \leq 24$ le coût $c(n)$ par la méthode dichotomique peut-il être amélioré ?

1.4 L'algorithme d'Euclide

Le but est de calculer le pgcd ("plus grand commun diviseur") de deux entiers naturels a et b. Rappelons que, si l'on note :

$$\mathcal{D}(a) := \{m \in \mathbf{N} \mid m \text{ divise } a\}$$

l'ensemble des diviseurs de a, alors le pgcd de a et b est défini comme le plus grand élément de $\mathcal{D}(a) \cap \mathcal{D}(b)$. Cette définition ne tombe en défaut que si $a = b = 0$ car alors $\mathcal{D}(a) = \mathcal{D}(b) = \mathbf{N}$ et il n'y a pas de plus grand élément dans $\mathcal{D}(a) \cap \mathcal{D}(b)$; nous conviendrons que le pgcd de 0 et 0 est 0. Nous noterons $a \wedge b$ le pgcd de a et b. On a les règles : $a \wedge b = b \wedge a$ et $a \wedge 0 = a$. Si a et b sont tous deux non nuls, on peut les décomposer en facteurs premiers :

$$a = p_1^{r_1} \cdots p_k^{r_k},$$
$$b = p_1^{s_1} \cdots p_k^{s_k},$$

où l'on a mis tous les facteurs premiers qui apparaissent dans a ou dans b, quitte à poser $r_i := 0$ si p_i ne divise que b, resp. $s_i := 0$ si p_i ne divise que a. On a alors :

$$a \wedge b = p_1^{\min(r_1,s_1)} \cdots p_k^{\min(r_k,s_k)}.$$

Malheureusement, *cette méthode est impraticable* car *il n'existe aucun algorithme raisonnablement rapide pour déterminer la décomposition en facteurs premiers d'un entier*. Il n'est pas question de justifier ici cette affirmation [2], il faudra donc l'admettre ! En revanche, il existe des algorithmes raisonnables pour calculer le pgcd.

1.4.1 L'algorithme de base

Il s'agit d'un algorithme rudimentaire mais élégant, qui remonte à Euclide. Il repose sur l'idée suivante :

Lemme 1.4.1 *Si $0 < b \leq a$, alors $a \wedge b = (a-b) \wedge b$.*

Preuve. - Si m divise a et b, alors il divise leur différence, d'où :

$$\mathcal{D}(a) \cap \mathcal{D}(b) \subset \mathcal{D}(a-b) \Longrightarrow \mathcal{D}(a) \cap \mathcal{D}(b) \subset \mathcal{D}(a-b) \cap \mathcal{D}(b).$$

2. Elle est liée à la théorie de la complexité, laquelle intervient dans l'une des sept "questions à un million de dollars" posées par l'institut Clay en l'an 2000. Rappelons que l'une des questions (mais pas celle-ci) a été résolue récemment par le russe Gregory Perelman.

Si m divise $a-b$ et b, alors il divise leur somme, d'où :

$$\mathcal{D}(a-b)\cap\mathcal{D}(b)\subset\mathcal{D}(a)\Longrightarrow\mathcal{D}(a-b)\cap\mathcal{D}(b)\subset\mathcal{D}(a)\cap\mathcal{D}(b).$$

On a donc $\mathcal{D}(a)\cap\mathcal{D}(b)=\mathcal{D}(a-b)\cap\mathcal{D}(b)$, d'où la conclusion. □

Pour calculer le pgcd de a et b, on peut donc, si $0<b\leq a$, les remplacer par $a-b$ et b, donc par des nombres plus petits. Si $b=0$, on a $a\wedge b=a$. Si $b>a$, on inverse les rôles, puisque $a\wedge b=b\wedge a$. Voici l'algorithme qui exprime cette description informelle :

```
x := a;
y := b;
tant que (x > 0 et y > 0)
  si (x < y) alors
     y := y - x;
  sinon
     x := x - y;
  fin si;
fin tant que;
si (x = 0) alors
   rendre (y);
sinon
   rendre (x);
fin si;
```

La terminaison est garantie parce qu'à chaque étape $x+y$ diminue strictement. L'invariant de boucle qui permet de prouver la correction découle du lemme : à chaque étape, on a $x\wedge y=a\wedge b$ et $x,y\geq 0$. En sortie de boucle, on a soit $x=0$ d'où $x\wedge y=y$, soit $y=0$ d'où $x\wedge y=x$.

Nous n'analyserons pas le coût de cet algorithme, qui n'est pas très facile à calculer, ni même à énoncer. On peut montrer que le cas le pire est celui qui est traité dans l'exercice suivant.

Exercice 1.4.2 On rappelle que la suite des nombres de Fibonacci est définie par les conditions initiales : $F_0:=0$ et $F_1:=1$; et par la relation de récurrence à deux pas : $F_{n+1}:=F_n+F_{n-1}$. Appliquer l'algorithme ci-dessus à $a:=F_{n+1}$ et $b:=F_n$.

TP 1.4.3 Programmer cet algorithme.

1.4.2 L'algorithme classique

Dans l'algorithme de base, si b est beaucoup plus petit que a, on remplace a par $a-b$ un grand nombre de fois, ce qui revient en fait à effectuer une division euclidienne de a par b. La version classique de l'algorithme d'Euclide incorpore directement cette division euclidienne :

```
x := a;
y := b;
tant que (y > 0)
   (q,r) := divEucl(x,y);
```

```
   x := y;
   y := r;
fin tant que;
rendre (x);;
```

A chaque étape, y diminue strictement (il est remplacé par le reste de la division euclidienne par y), ce qui garantit la terminaison de l'algorithme. Par ailleurs, l'invariant de boucle est : $y \geq 0$ et $x \wedge y = a \wedge b$ (pour la preuve, voir l'exercice ci-dessous) qui donne en sortie $x = x \wedge 0 = a \wedge b$, d'où la correction. L'analyse du coût est difficile, mais on peut montrer que le cas le pire est le même que dans la version de base (avec les nombres de Fibonacci).

Exercice 1.4.4 Si $(q,r) := \text{diveucl}(x,y)$, montrer que $x \wedge y = y \wedge r$. En déduire la validité de l'invariant de boucle ci-dessus.

TP 1.4.5 Programmer cet algorithme.

1.4.3 L'algorithme amélioré

A chaque étape de l'algorithme, x et y sont des "combinaisons linéaires à coefficients entiers relatifs de a et b", autrement dit, on peut écrire $x = sa + tb$ et $y = ua + vb$, avec $s,t,u,v \in \mathbf{Z}$. C'est évident au départ, puisqu'ils sont initialisés avec les valeurs $x := a = 1.a + 0.b$ et $y := b = 0.a + 1.b$. Cela reste vrai au cours de l'exécution de l'algorithme, puisque x est remplacé par y et que y est remplacé par $x - qy$ avec $q \in \mathbf{Z}$. Comme à la fin de l'algorithme on a $x = a \wedge b$, on en déduit le *théorème de Bézout : il existe des entiers relatifs s,t tels que $a \wedge b = sa + tb$*. Naturellement, on ne peut espérer trouver s et t naturels puisque le pgcd est plus petit que a et que b.

L'algorithme suivant exploite cette idée pour calculer les "coefficients de Bézout" s,t en même temps que le pgcd :

```
x := a;
y := b;
s := 1;  t := 0;
u := 0;  v := 1;
tant que y > 0
   (q,r) := divEucl(x,y);
   x := y;
   y := r,
   w := u;  u := s - q*u;  s := w;
   w := v;  v := t - q*v;  t := w;
fin tant que;
rendre (x,s,t);;
```

La terminaison se prouve comme ci-dessus. Pour prouver la correction, on ajoute à l'invariant de boucle les égalités $x = sa + tb$ et $y = ua + vb$.

Exercice 1.4.6 A quoi sert la variable w dans cet algorithme ?

TP 1.4.7 Programmer cet algorithme.

1.5 Suppléments facultatifs

Résolution de l'équation $ax + by = c$.

Nombres premiers. Le crible d'Eratosthène.

Chapitre 2

Arithmétique modulaire et applications

2.1 Congruences

2.1.1 L'ensemble $\mathbf{Z}/m\mathbf{Z}$

On fixe un entier naturel $m \geq 2$.

Définition 2.1.1 Soient $a, b \in \mathbf{Z}$. On dit que *a est congru à b modulo m* si $b - a$ est multiple de m ; on écrit $a \equiv b \pmod{m}$:

$$a \equiv b \pmod{m} \Longleftrightarrow \exists k \in \mathbf{Z} \; : \; b - a = km.$$

Proposition 2.1.2 *Pour que les entiers a et b soient congrus modulo m, il faut, et il suffit, que leurs divisions euclidiennes par m donnent le même reste.*

Preuve. - Si $a = qm + r$ et si $b = q'm + r$, alors $b - a = km$ avec $k = q' - q$, donc $a \equiv b \pmod{m}$. Réciproquement, supposons que $a \equiv b \pmod{m}$ et soit $k \in \mathbf{Z}$ tel que $b - a = km$. Si la division euclidienne de a par m s'écrit $a = qm + r$, $0 \leq r \leq m - 1$, alors $b = (q + k)m + r$, de sorte que r est le reste de la division euclidienne de b par m. □

La relation de congruence modulo m vérifie les trois propriétés suivantes :

Réflexivité : Quel que soit $a \in \mathbf{Z}$, on a $a \equiv a \pmod{m}$.

Symétrie : Quels que soient $a, b \in \mathbf{Z}$, si $a \equiv b \pmod{m}$, alors $b \equiv a \pmod{m}$.

Transitivité : Quels que soient $a, b, c \in \mathbf{Z}$, si $a \equiv b \pmod{m}$ et si $b \equiv c \pmod{m}$, alors $a \equiv c \pmod{m}$.

On résume ces propriétés en disant que la relation de congruence modulo m est une *relation d'équivalence*. Cela permet de regrouper les éléments de $\mathbf{Z}$ en *classes d'équivalence*, également appelées dans ce cas *classes de congruence* (on ne précise pas toujours le "module" m) : on dit que a et b sont dans la même classe s'ils sont congrus modulo m.

Exemple 2.1.3 Dans le cas où $m = 2$, la relation de congruence est la relation "a et b ont même parité". Il y a deux classes d'équivalence : celle des entiers pairs et celle des entiers impairs.

On notera $a \pmod{m}$ ou $\overline{a}$ la classe de a ; dans le second cas, la notation ne précise pas le module m, mais on espère que cela ne causera pas de confusion. La principale règle à retenir est la suivante :

$$\boxed{\overline{a} = \overline{b} \Longleftrightarrow a \equiv b \pmod{m}}$$

Remarque 2.1.4 Ce n'est pas n'importe quelle relation entre entiers qui permettrait de les regrouper en classes ! Par exemple, aucune des relations "a est différent de b", "a est supérieur à b", "a divise b", n'est une relation d'équivalence et ces relations ne permettent pas un tel regroupement.

Bien que l'ensemble $\mathbf{Z}$ des entiers relatifs soit infini, il n'y a qu'un nombre fini de classes de congruence. Si $m = 2$, on a vu qu'il y en avait deux. Si $m = 3$, il y a la classe de 0, notée $\overline{0}$; puis il y a la classe de 1, notée $\overline{1}$; puis il y a celle de 2, notée $\overline{2}$; puis celle de 3, notée $\overline{3}$ ... mais c'est la même que celle de 0 puisque $3 \equiv 0 \pmod 3$. De même, on constate que $\overline{4} = \overline{1}$, que $\overline{5} = \overline{2}$, etc. Si l'on part "à gauche" de 0, rien de neuf : $\overline{-1} = \overline{2}$ puisque $-1 \equiv 2 \pmod 3$. En fin de compte, il n'y a que trois classes : $\overline{0}$, $\overline{1}$ et $\overline{2}$. Il y en a bien trois car par exemple $\overline{0} \neq \overline{1}$ puisque $1 \not\equiv 0 \pmod 3$.

Proposition 2.1.5 *(i) Pour tout $a \in \mathbf{Z}$, il existe un unique $b \in \{0, \ldots, m-1\}$ tel que $\overline{a} = \overline{b}$.*
(ii) Il y a exactement m classes de congruence modulo m, qui sont $\overline{0}, \overline{1}, \ldots, \overline{m-1}$.

Preuve. - (i) L'unique b dont on affirme l'existence est tout simplement le reste de la division euclidienne de a par m.
(ii) C'est une conséquence immédiate de (i). □

On notera $\mathbf{Z}/m\mathbf{Z}$ l'ensemble des classes de congruence modulo m. On a donc :

$$\boxed{\mathbf{Z}/m\mathbf{Z} = \{\overline{0}, \overline{1}, \ldots, \overline{m-1}\}}$$

Par conséquent :

$$\text{card } \mathbf{Z}/m\mathbf{Z} = m.$$

Pratiquement, si l'on veut travailler avec un élément x de $\mathbf{Z}/m\mathbf{Z}$, c'est-à-dire avec une classe de congruence modulo m, on choisit un entier $a \in \mathbf{Z}$ dont x est la classe : $x = \overline{a}$, et l'on travaille avec a (voir un peu plus bas l'exemple de l'addition des classes de congruence). On dit que *a est un représentant de la classe x.*

Exemple 2.1.6 On a $\mathbf{Z}/2\mathbf{Z} = \{\overline{0}, \overline{1}\}$. Les entiers -8 et 39 sont des représentants respectifs de ces deux classes.
On a $\mathbf{Z}/3\mathbf{Z} = \{\overline{0}, \overline{1}, \overline{2}\}$. Les entiers 39 et -8 sont des représentants respectifs des deux premières classes, l'entier 2000 est un représentant de la troisième.

Remarque 2.1.7 La notation est ambigue, puisque l'on ne précise pas le module. Ainsi, dans $\mathbf{Z}/2\mathbf{Z}$, on a $\overline{0} = \overline{2}$ alors que c'est faux dans $\mathbf{Z}/3\mathbf{Z}$.

2.1.2 Le groupe $(\mathbf{Z}/m\mathbf{Z}, +)$

Proposition 2.1.8 *Soient $a, a', b, b' \in \mathbf{Z}$. On suppose : $a \equiv a' \pmod m$ et $b \equiv b' \pmod m$. Alors $a + b \equiv a' + b' \pmod m$.*

Preuve. - Si $a' - a = km$ et $b' - b = lm$, alors $(a' + b') - (a + b) = (k + l)m$. □

Cette propriété est appelée *compatibilité de la relation de congruence avec l'addition*. On en déduit immédiatement :

Corollaire 2.1.9 *Soient $a, a', b, b' \in \mathbf{Z}$. On suppose : $\overline{a} = \overline{a'}$ et $\overline{b} = \overline{b'}$. Alors $\overline{a + b} = \overline{a' + b'}$.*

Ce corollaire permet de définir une "loi de composition interne", en l'occurrence une addition, sur l'ensemble $\mathbf{Z}/m\mathbf{Z}$ des classes de congruence modulo m. On procède de la façon suivante. Soient x et y deux éléments de $\mathbf{Z}/m\mathbf{Z}$. On choisit un représentant $a \in \mathbf{Z}$ de x et un représentant $b \in \mathbf{Z}$ de y. Alors, si $c := a+b$, on pose $x+y := \overline{c}$. Cette définition pose un problème : obtiendrait-on le même résultat si l'on avait choisi d'autres représentants ? (Pour chaque classe, il y a une infinité de représentants !) Pour le savoir, on choisit un autre représentant $a' \in \mathbf{Z}$ de x et un autre représentant $b' \in \mathbf{Z}$ de y, et l'on calcule $c' := a'+b'$. Logiquement, l'application de la "définition" ci-dessus donne tout aussi bien $x+y := \overline{c'}$. Pour que ces deux calculs soient cohérents, il faut être absolument certain que $\overline{c} = \overline{c'}$. heureusement, le corollaire ci-dessus nous le garantit. En effet, $\overline{a} = \overline{a'}$ (les deux sont égaux à x) et $\overline{b} = \overline{b'}$ (les deux sont égaux à y), donc, en vertu du corollaire, $\overline{a+b} = \overline{a'+b'}$, c'est-à-dire justement $\overline{c} = \overline{c'}$. L'addition sur $\mathbf{Z}/m\mathbf{Z}$ est donc bien définie. Son mode d'emploi tient principalement dans la règle suivante, qui traduit sa définition :

$$\boxed{\overline{a}+\overline{b} = \overline{a+b}}$$

Voici à titre d'exemple les tables d'addition modulo m pour $m = 2,3,4,5,6$:

$+$	$\overline{0}$	$\overline{1}$
$\overline{0}$	$\overline{0}$	$\overline{1}$
$\overline{1}$	$\overline{1}$	$\overline{0}$

$+$	$\overline{0}$	$\overline{1}$	$\overline{2}$
$\overline{0}$	$\overline{0}$	$\overline{1}$	$\overline{2}$
$\overline{1}$	$\overline{1}$	$\overline{2}$	$\overline{0}$
$\overline{2}$	$\overline{2}$	$\overline{0}$	$\overline{1}$

$+$	$\overline{0}$	$\overline{1}$	$\overline{2}$	$\overline{3}$
$\overline{0}$	$\overline{0}$	$\overline{1}$	$\overline{2}$	$\overline{3}$
$\overline{1}$	$\overline{1}$	$\overline{2}$	$\overline{3}$	$\overline{0}$
$\overline{2}$	$\overline{2}$	$\overline{3}$	$\overline{0}$	$\overline{1}$
$\overline{3}$	$\overline{3}$	$\overline{0}$	$\overline{1}$	$\overline{2}$

$+$	$\overline{0}$	$\overline{1}$	$\overline{2}$	$\overline{3}$	$\overline{4}$
$\overline{0}$	$\overline{0}$	$\overline{1}$	$\overline{2}$	$\overline{3}$	$\overline{4}$
$\overline{1}$	$\overline{1}$	$\overline{2}$	$\overline{3}$	$\overline{4}$	$\overline{0}$
$\overline{2}$	$\overline{2}$	$\overline{3}$	$\overline{4}$	$\overline{0}$	$\overline{1}$
$\overline{3}$	$\overline{3}$	$\overline{4}$	$\overline{0}$	$\overline{1}$	$\overline{2}$
$\overline{4}$	$\overline{4}$	$\overline{0}$	$\overline{1}$	$\overline{2}$	$\overline{3}$

$+$	$\overline{0}$	$\overline{1}$	$\overline{2}$	$\overline{3}$	$\overline{4}$	$\overline{5}$
$\overline{0}$	$\overline{0}$	$\overline{1}$	$\overline{2}$	$\overline{3}$	$\overline{4}$	$\overline{5}$
$\overline{1}$	$\overline{1}$	$\overline{2}$	$\overline{3}$	$\overline{4}$	$\overline{5}$	$\overline{0}$
$\overline{2}$	$\overline{2}$	$\overline{3}$	$\overline{4}$	$\overline{5}$	$\overline{0}$	$\overline{1}$
$\overline{3}$	$\overline{3}$	$\overline{4}$	$\overline{5}$	$\overline{0}$	$\overline{1}$	$\overline{2}$
$\overline{4}$	$\overline{4}$	$\overline{5}$	$\overline{0}$	$\overline{1}$	$\overline{2}$	$\overline{3}$
$\overline{5}$	$\overline{5}$	$\overline{0}$	$\overline{1}$	$\overline{2}$	$\overline{3}$	$\overline{4}$

L'addition dans $\mathbf{Z}/m\mathbf{Z}$ vérifie les propriétés suivantes :

Commutativité : Pour tous $x,y \in \mathbf{Z}/m\mathbf{Z}$, on a $x+y = y+x$.

Associativité : Pour tous $x,y,z \in \mathbf{Z}/m\mathbf{Z}$, on a $(x+y)+z = x+(y+z)$.

Elément neutre : Pour tout $x \in \mathbf{Z}/m\mathbf{Z}$, on a $x+\overline{0} = \overline{0}+x = x$.

Symétrique : Si $x = \overline{a}$, l'élément $x' := \overline{-a}$ vérifie : $x+x' = x'+x = \overline{0}$.

On résume ces propriétés en disant que l'ensemble $\mathbf{Z}/m\mathbf{Z}$ muni de l'addition est un *groupe commutatif*. Pratiquement, cela signifie que le calcul avec les classes de congruence ressemble beaucoup au calcul sur les nombres usuels. Il y a bien quelques différences auxquelles il faut prendre garde ; on les mentionnera à la volée.

Revenons sur le symétrique. Il est facile de prouver que si $\overline{a} = \overline{a'}$, alors $\overline{-a} = \overline{-a'}$. On peut donc poser $-\overline{a} := \overline{-a}$. Si $x = \overline{a}$, l'élément x' de la dernière propriété ci-dessus est tout simplement $x' = \overline{-a}$: c'est *l'opposé* de x. Son existence permet de définir $y-x := y+(-x)$: c'est l'unique z tel que $x+z = y$. On a les règles pratiques :

$$\boxed{-\overline{a} = \overline{-a} \text{ et } \overline{a}-\overline{b} = \overline{a-b}}$$

2.1.3 L'anneau $(\mathbf{Z}/m\mathbf{Z},+,\times)$

Nous allons définir une multiplication sur $\mathbf{Z}/m\mathbf{Z}$ selon un processus très similaire à celui de l'addition : nous irons donc un peu plus vite.

Proposition 2.1.10 *Soient $a,a',b,b' \in \mathbf{Z}$. On suppose : $a \equiv a' \pmod{m}$ et $b \equiv b' \pmod{m}$. Alors $ab \equiv a'b' \pmod{m}$.*

Preuve. - Si $a'-a = km$ et $b'-b = lm$, alors $a'b'-ab = (la+kb+klm)m$. □

La relation de congruence est donc compatible avec la multiplication, d'où :

Corollaire 2.1.11 *Soient $a,a',b,b' \in \mathbf{Z}$. On suppose : $\overline{a} = \overline{a'}$ et $\overline{b} = \overline{b'}$. Alors $\overline{ab} = \overline{a'b'}$.*

Cela permet de définir la multiplication sur $\mathbf{Z}/m\mathbf{Z}$ de la façon suivante. Soient x et y deux éléments de $\mathbf{Z}/m\mathbf{Z}$. On choisit un représentant $a \in \mathbf{Z}$ de x et un représentant $b \in \mathbf{Z}$ de y. Alors, si $c := ab$, on pose $xy := \overline{c}$. Cette définition est cohérente en vertu du corollaire. On a la règle :

$$\boxed{\overline{a}\,\overline{b} = \overline{ab}}$$

Voici à titre d'exemple les tables de multiplication modulo m pour $m = 2,3,4,5,6$:

$\times$	$\overline{0}$	$\overline{1}$
$\overline{0}$	$\overline{0}$	$\overline{0}$
$\overline{1}$	$\overline{0}$	$\overline{1}$

$\times$	$\overline{0}$	$\overline{1}$	$\overline{2}$
$\overline{0}$	$\overline{0}$	$\overline{0}$	$\overline{0}$
$\overline{1}$	$\overline{0}$	$\overline{1}$	$\overline{2}$
$\overline{2}$	$\overline{0}$	$\overline{2}$	$\overline{1}$

$\times$	$\overline{0}$	$\overline{1}$	$\overline{2}$	$\overline{3}$
$\overline{0}$	$\overline{0}$	$\overline{0}$	$\overline{0}$	$\overline{0}$
$\overline{1}$	$\overline{0}$	$\overline{1}$	$\overline{2}$	$\overline{3}$
$\overline{2}$	$\overline{0}$	$\overline{2}$	$\overline{0}$	$\overline{2}$
$\overline{3}$	$\overline{0}$	$\overline{3}$	$\overline{2}$	$\overline{1}$

$\times$	$\overline{0}$	$\overline{1}$	$\overline{2}$	$\overline{3}$	$\overline{4}$
$\overline{0}$	$\overline{0}$	$\overline{0}$	$\overline{0}$	$\overline{0}$	$\overline{0}$
$\overline{1}$	$\overline{0}$	$\overline{1}$	$\overline{2}$	$\overline{3}$	$\overline{4}$
$\overline{2}$	$\overline{0}$	$\overline{2}$	$\overline{4}$	$\overline{1}$	$\overline{3}$
$\overline{3}$	$\overline{0}$	$\overline{3}$	$\overline{1}$	$\overline{4}$	$\overline{2}$
$\overline{4}$	$\overline{0}$	$\overline{4}$	$\overline{3}$	$\overline{2}$	$\overline{1}$

$\times$	$\overline{0}$	$\overline{1}$	$\overline{2}$	$\overline{3}$	$\overline{4}$	$\overline{5}$
$\overline{0}$	$\overline{0}$	$\overline{0}$	$\overline{0}$	$\overline{0}$	$\overline{0}$	$\overline{0}$
$\overline{1}$	$\overline{0}$	$\overline{1}$	$\overline{2}$	$\overline{3}$	$\overline{4}$	$\overline{5}$
$\overline{2}$	$\overline{0}$	$\overline{2}$	$\overline{4}$	$\overline{0}$	$\overline{2}$	$\overline{4}$
$\overline{3}$	$\overline{0}$	$\overline{3}$	$\overline{0}$	$\overline{3}$	$\overline{0}$	$\overline{3}$
$\overline{4}$	$\overline{0}$	$\overline{4}$	$\overline{2}$	$\overline{0}$	$\overline{4}$	$\overline{2}$
$\overline{5}$	$\overline{0}$	$\overline{5}$	$\overline{4}$	$\overline{3}$	$\overline{2}$	$\overline{1}$

La multiplication dans $\mathbf{Z}/m\mathbf{Z}$ vérifie les propriétés suivantes :

Commutativité : Pour tous $x,y \in \mathbf{Z}/m\mathbf{Z}$, on a $xy = yx$.

Associativité : Pour tous $x,y,z \in \mathbf{Z}/m\mathbf{Z}$, on a $(xy)z = x(yz)$.

Elément neutre : Pour tout $x \in \mathbf{Z}/m\mathbf{Z}$, on a $x\overline{1} = \overline{1}x = x$.

Distributivité : Pour tous $x,y \in \mathbf{Z}/m\mathbf{Z}$, on a $x(y+z) = xy+xz$ et $(y+z)x = yx+zx$.

Ajoutons que $\overline{0}$ est un élément *absorbant* : pour tout $x \in \mathbf{Z}/m\mathbf{Z}$, on a $x\overline{0} = \overline{0}x = \overline{0}$. On résume ces propriétés en disant que $\mathbf{Z}/m\mathbf{Z}$ muni de l'addition et de la multiplication est un *anneau commutatif*.

Il y a cependant deux propriétés qui manquent à l'appel : l'*intégrité* et l'existence d'un *inverse*. Par exemple, dans $\mathbf{Z}/4\mathbf{Z}$, l'élément $\overline{2}$ n'est pas nul, mais $\overline{2}\times\overline{2} = \overline{0}$. De plus, aucun élément $x \in \mathbf{Z}/4\mathbf{Z}$ ne vérifie $\overline{2}\times x = \overline{1}$. De même, dans $\mathbf{Z}/6\mathbf{Z}$, les éléments $\overline{2}$ et $\overline{3}$ ne sont pas nuls, mais $\overline{2}\times\overline{3} = \overline{0}$. De plus, aucun élément $x \in \mathbf{Z}/6\mathbf{Z}$ ne vérifie $\overline{2}\times x = \overline{1}$, ni d'ailleurs $\overline{3}\times x = \overline{1}$. En revanche, dans $\mathbf{Z}/2\mathbf{Z}$, $\mathbf{Z}/3\mathbf{Z}$ et $\mathbf{Z}/5\mathbf{Z}$, on vérifie par simple examen des tables de multiplication que le produit de deux éléments non nuls est non nul (on dit que ces anneaux sont *intègres*) ; et

que tout élément non nul x admet un inverse x', c'est-à-dire que $x \times x' = x' \times x = \overline{1}$ (on dit que ces anneaux sont des *corps*). La différence la plus visible entre $4,6$ d'une part, $2,3,5$ d'autre part est que chacun de ces derniers est *premier*, *i.e.* il n'admet pas d'autre diviseur que lui-même et 1. Nous allons voir que c'est bien la raison de cette différence de propriétés de la multiplication.

Nous dirons que $x \in \mathbf{Z}/m\mathbf{Z}$ est *diviseur de zéro* s'il existe $y \neq \overline{0}$ tel que $xy = \overline{0}$; et que x est *inversible* s'il existe $x' \in \mathbf{Z}/m\mathbf{Z}$ tel que $xx' = \overline{1}$.

Théorème 2.1.12 *(i) Si m est premier, l'anneau $\mathbf{Z}/m\mathbf{Z}$ est intègre (autrement dit, $\overline{0}$ est le seul diviseur de zéro) ; et c'est un corps (autrement dit, tout $x \neq \overline{0}$ est inversible).*
(ii) Si m n'est pas premier, l'anneau $\mathbf{Z}/m\mathbf{Z}$ n'est ni intègre ni un corps.

Preuve. - C'est en fait un cas particulier de la proposition 2.1.14 que nous prouverons au paragraphe suivant. □

L'exemple le plus fondamental est celui où $m = 2$. Le corps $\mathbf{Z}/2\mathbf{Z}$ est parfois noté $\mathbf{F}_2$.

TP 2.1.13 Programmer le calcul dans l'anneau $\mathbf{Z}/m\mathbf{Z}$, y compris le calcul de l'inverse d'un élément quand m est premier.

2.1.4 Le groupe $\left((\mathbf{Z}/m\mathbf{Z})^*, \times\right)$

Proposition 2.1.14 *Soit $a \in \{0, \ldots, m-1\}$ et soit $d := a \wedge m$ le pgcd de a et m.*
(i) Si $d > 1$, alors $\overline{a}$ est un diviseur de zéro et n'est pas inversible dans $\mathbf{Z}/m\mathbf{Z}$.
(ii) Si $d = 1$, alors $\overline{a}$ n'est pas un diviseur de zéro et $\overline{a}$ est inversible dans $\mathbf{Z}/m\mathbf{Z}$.

Preuve. - (i) On écrit $a = da'$ et $m = dm'$. Puisque $d > 1$, on a $0 < m' < m$ donc $m' \not\equiv 0 \pmod m$ donc $\overline{m'} \neq \overline{0}$. Par ailleurs :

$$\overline{a}\,\overline{m'} = \overline{am'} = \overline{da'm'} = \overline{a'm} = \overline{0}.$$

Comme $\overline{m'} \neq 0$, on en déduit que $\overline{a}$ est un diviseur de zéro. D'autre part, si $\overline{a}$ était inversible, on pourrait écrire $\overline{a}\overline{b} = \overline{1}$, d'où $ab = 1 + km$, d'où $d(a'b - km') = 1$, ce qui est impossible puisque $d > 1$.
(ii) D'après le théorème de Bézout, il existe $k, b \in \mathbf{Z}$ tels que $ab - km = 1$. Alors $\overline{a}\overline{b} = \overline{1}$ et $\overline{a}$ est bien inversible dans $\mathbf{Z}/m\mathbf{Z}$. Supposons alors que $\overline{a}x = \overline{0}$. En multipliant les deux membres de cette égalité par $\overline{b}$, on trouve que $x = \overline{0}$, ce qui montre que $\overline{a}$ n'est pas un diviseur de zéro. □

On est conduit à distinguer les éléments inversibles de $\mathbf{Z}/m\mathbf{Z}$, c'est-à-dire (d'après la proposition) les éléments de la forme $\overline{a}$, où $a \in \{0, \ldots, m-1\}$ et où $a \wedge m = 1$ (autrement dit, a et m sont *premiers entre eux*). On note $(\mathbf{Z}/m\mathbf{Z})^*$ l'ensemble de ces éléments et $\phi(m) := \text{card}(\mathbf{Z}/m\mathbf{Z})^*$ le nombre de ces éléments. La fonction ϕ est *l'indicatrice d'Euler*.

En examinant, les lignes (ou les colonnes) où figure $\overline{1}$ les tables de multiplication, on trouve par exemple : $(\mathbf{Z}/2\mathbf{Z})^* = \{\overline{1}\}$, d'où $\phi(2) = 1$; $(\mathbf{Z}/3\mathbf{Z})^* = \{\overline{1}, \overline{2}\}$, d'où $\phi(3) = 2$; $(\mathbf{Z}/4\mathbf{Z})^* = \{\overline{1}, \overline{3}\}$, d'où $\phi(4) = 2$; $(\mathbf{Z}/5\mathbf{Z})^* = \{\overline{1}, \overline{2}, \overline{3}, \overline{4}\}$, d'où $\phi(5) = 4$; et $(\mathbf{Z}/6\mathbf{Z})^* = \{\overline{1}, \overline{5}\}$, d'où $\phi(6) = 2$.

Si m est premier, il est clair que $(\mathbf{Z}/m\mathbf{Z})^* = \{\overline{1}, \ldots, \overline{m-1}\}$, d'où $\phi(m) = m - 1$.

Le théorème qui suit est d'une importance cruciale dans les deux applications que nous avons en vue : générateurs pseudo-aléatoires (voir la section 2.2) et méthode RSA de cryptographie à clé publique (voir la section 2.3).

Théorème 2.1.15 *(i) Pour tout $x \in (\mathbf{Z}/m\mathbf{Z})^*$, on a $x^{\phi(m)} = \overline{1}$.*
(ii) Pour tout $a \in \mathbf{Z}$ tel que $a \wedge m = 1$, on a $a^{\phi(m)} \equiv 1 \pmod{m}$.
(iii) Si m est premier et si $a \in \mathbf{Z}$ n'est pas multiple de m, alors $a^{m-1} \equiv 1 \pmod{m}$.

Preuve. - (i) C'est un cas particulier du théorème de Lagrange, qui sera démontré dans le cours d'algèbre de L2.
(ii) Ce théorème, dû à Euler, est simplement la traduction de (i) en termes de congruences.
(iii) Ce cas particulier de (ii) est le *petit théorème de Fermat*. □

Exercice 2.1.16 Si $m = p^r$, où p est premier, alors $\phi(m) = p^r - p^{r-1}$. Si $m = pq$, où p et q sont premiers distincts, alors $\phi(m) = (p-1)(q-1)$.

TP 2.1.17 Programmer, pour tout m, la vérification du théorème 2.1.15 : recherche de tous les éléments de $(\mathbf{Z}/m\mathbf{Z})^*$, calcul de $\phi(m)$ et calcul de $x^{\phi(m)}$ pour $x \in (\mathbf{Z}/m\mathbf{Z})^*$.

2.2 Générateurs pseudo-aléatoires

L'un des pères fondateurs de l'informatique, John von Neumann, a dit : *"Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin."* [1]. Nous tenterons donc plus modestement de *simuler* l'aléatoire. Dans un premier paragraphe, nous décrirons des moyens de produire de longues suites de nombres variés ; et dans un deuxième paragraphe, des moyens de tester si ces suites ont l'air aléatoire. Il y a, pour ces deux phases, une théorie riche et compliquée (et amusante). Nous ne l'effleurerons même pas : il n'y a ici que quelques recettes faciles à mettre en oeuvre.

Vu de l'extérieur, un générateur (pseudo)-aléatoire est un programme qui rend un nombre chaque fois qu'on l'appelle. Vu de l'intérieur, ce programme calcule en fait les termes d'une suite numérique (x_n). Chaque fois qu'on l'appelle, il renvoie un terme ; d'un appel sur l'autre, il passe au terme suivant. Autrement dit, entre deux appels, le programme garde trace du rang du dernier terme rendu : l'indice n est une *variable rémanente*, et il faut donc savoir programmer de telles variables. Le plus souvent, il s'agit de suites définies par récurrence : $x_{n+1} = f(x_n)$. Dans ce cas, la variable rémanente contient la dernière valeur de x_n qui a été rendue. Dans notre approche élémentaire, nous ne traiterons pas ce problème : nous considèrerons simplement des programmes qui calculent $f(x)$, donc chaque nouveau terme en fonction du précédent.

Il existe deux types principaux de générateurs aléatoires : ceux qui rendent des nombres réels entre 0 et 1 ; et ceux qui rendent des entiers entre 0 et $m-1$, l'entier $m \geq 2$ ayant été passé en argument. Nous ne nous occuperons que du deuxième type.

Exercice 2.2.1 Comment utiliser un générateur de l'un des deux types pour émuler l'autre ?

1. "Quiconque envisage des méthodes arithmétiques pour produire des nombres aléatoires est, bien entendu, en état de péché."

2.2.1 Générateurs linéaires congruentiels

Nous supposons fixé le *module* $m \geq 2$ (argument d'appel). Au lieu de considérer une suite d'entiers compris entre 0 et $m-1$, nous considèrerons une suite d'éléments de $\mathbf{Z}/m\mathbf{Z}$. Le format général de cette suite est donné par la relation de récurrence :

$$\forall n \in \mathbf{N}\ ,\ x_{n+1} := ax_n + b.$$

Il faut donc choisir $a, b, x_0 \in \mathbf{Z}/m\mathbf{Z}$. On ne prendra évidemment pas $a := \overline{0}$ car la suite serait constante à partir du rang 1 et n'aurait pas du tout l'air aléatoire. On ne prendra pas non plus en général $a = \overline{1}$, car on aurait $x_n = x_0 + nb$, et l'on repèrerait trop facilement une régularité. (Attention : la notation nb désigne ici la somme de n termes égaux à b avec l'addition de $\mathbf{Z}/m\mathbf{Z}$.)

Cas d'un module premier

Supposons m premier. Puisque l'on a exclu le cas où $a = \overline{1}$, l'élément $\overline{1} - a$ est inversible dans $\mathbf{Z}/m\mathbf{Z}$ et l'on va pouvoir appliquer la méthode vue en terminale pour les suites arithmético-géométriques. On cherche un "point fixe" u :

$$u = au + b \Longleftrightarrow (\overline{1} - a)u = b \Longleftrightarrow u = b(\overline{1} - a)^{-1},$$

où l'on note x^{-1} l'inverse d'un élément (inversible) de $\mathbf{Z}/m\mathbf{Z}$. La relation de récurrence devient alors :

$$x_{n+1} := ax_n + b \Longleftrightarrow (x_{n+1} - u) := a(x_n - u),$$

et l'on en déduit $x_n - u = a^n(x_0 - u)$, donc :

$$\forall n \in \mathbf{N}\ ,\ x_n = a^n(x_0 - u) + u.$$

(Attention : la notation a^n désigne ici le produit de n facteurs égaux à a avec la multiplication de $\mathbf{Z}/m\mathbf{Z}$.) Finalement, à un décalage près de valeur u, tout se passe comme si la suite était géométrique. Pour la suite de ce paragraphe, nous supposons donc que $b = u = \overline{0}$ et donc que $x_n = a^n x_0$.

On prendra évidemment $x_0 \neq \overline{0}$, sinon la suite serait constante et n'aurait pas du tout l'air aléatoire. Finalement, on est donc conduit à examiner la suite des $a^n x_0$. Pour savoir si elle a l'air aléatoire, nous décrirons plus loin des tests empiriques. On peut déjà se demander si elle ne boucle pas trop vite. En effet, le petit théorème de Fermat nous dit que $a^{m-1} = \overline{1}$ et donc que, si $n = q(m-1) + r$, on a $a^n = (a^{m-1})^q a^r = a^r$, d'où $x_n = x_r$: cela signifie que la suite se reproduit avec une période $(m-1)$. Mais ce n'est pas nécessairement *la* période de cette suite.

Exemple 2.2.2 Prenons $m := 7$. Tout $a \in (\mathbf{Z}/7\mathbf{Z})^*$ vérifie $a^6 = \overline{1}$, donc la suite des $a^n x_0$ se reproduit à coup sûr avec une période 6. Mais si l'on prend par exemple $a := \overline{2}$, on a $a^3 = \overline{1}$ et la période tombe à 3. Il vaut mieux prendre $a := \overline{3}$ qui est tel que $a^0 = \overline{1}$, $a^1 = \overline{3}$, $a^2 = \overline{2}$, $a^3 = \overline{6}$, $a^4 = \overline{4}$, $a^5 = \overline{5}$ sont distincts et qui fournit donc une suite aussi longue que possible.

Dans tous les cas, voici ce dont est sûr quel que soit $a \in (\mathbf{Z}/m\mathbf{Z})^*$:

1. Il existe un plus petit entier $n \geq 1$ tel que $a^n = \overline{1}$.
2. Cet entier est un diviseur de $m-1$.

3. La suite (x_n) est de période n, elle admet n termes distincts.

L'entier n est appelé *ordre* de l'élément $a \in (\mathbf{Z}/m\mathbf{Z})^*$. Et maintenant, comme dans l'exemple ci-dessus, il y a moyen de choisir un élément d'ordre optimal :

Théorème 2.2.3 *Il existe un élément $a \in (\mathbf{Z}/m\mathbf{Z})^*$ d'ordre $m-1$.*

Preuve. - Ce sera prouvé dans le cours d'algèbre de L2. □

Cas d'un module primaire

Supposons *m primaire*, c'est-à-dire puissance d'un nombre premier : $n = p^r$, où p est premier et $r \geq 2$. Dans ce cas, il n'est pas aussi facile d'étudier une suite arithmético-géométrique générale, aussi prendrons nous d'emblée $b := \overline{0}$. Et, bien entendu, on suppose toujours $a \neq \overline{0}, \overline{1}$ et $x_0 \neq \overline{0}$.

Si a est la classe d'un multiple de p, alors a^r est la classe d'un multiple de p^r, donc $a^r = \overline{0}$. Dans ce cas, la suite est stationnaire en $\overline{0}$ et donc très peu aléatoire. Nous supposons donc que a est la classe d'un entier non multiple de p. Mais un tel entier n'a pas de diviseur commun avec $p^r = m$, donc a est inversible : on a encore $a \in (\mathbf{Z}/m\mathbf{Z})^*$. la situation est analogue à celle décrite plus haut :

1. Il existe un plus petit entier $n \geq 1$ tel que $a^n = \overline{1}$.
2. Cet entier est un diviseur de $\phi(m) = p^r - p^{r-1}$.
3. La suite (x_n) est de période n, elle admet n termes distincts.

On dit encore que n est l'*ordre* de a. En ce qui concerne le choix d'un élément d'ordre optimal, la situation est un tout petit peu plus compliquée :

Théorème 2.2.4 *(i) Si p est impair ou si m est égal à 2 ou 4, il existe un élément $a \in (\mathbf{Z}/m\mathbf{Z})^*$ d'ordre $\phi(m)$.*
(ii) Si $m = 2^r$, $r \geq 3$, on a $\phi(m) = 2^{r-1}$. Il n'existe pas d'élément d'ordre 2^{r-1} ; l'ordre optimal est 2^{r-2}. C'est l'ordre de l'élément $\overline{3}$ (mais pas uniquement de ce dernier).

Exercice 2.2.5 Démontrer que l'ordre d'un élément de $(\mathbf{Z}/m\mathbf{Z})^*$ divise $\phi(m)$.

TP 2.2.6 Trouver dans $(\mathbf{Z}/m\mathbf{Z})^*$ un élément d'ordre optimal.

2.2.2 Quelques tests

Le fait que le générateur pseudo-aléatoire ait une grande période n'est pas un critère suffisant de bonne qualité. Par exemple, pour $m = 2$, un tel générateur peut être assimilé à un simulateur de tirage à pile ou face (mettons que pile = 1 et face = 0). S'il tire avec régularité $(10^9 - 1)$ fois pile et une fois face, sa période sera 10^9 (ce qui n'est pas mal !) mais on le trouvera suspect. Nous donnons ici, sans justification autre que l'intuition, quelques tests faciles à comprendre et à mettre en oeuvre.

Tests empiriques de fréquence

La première vérification, inspirée de l'exemple ci-dessus, est celle des fréquences : dans une longue séquence de tirages successifs, chacune des valeurs possibles $0,\dots,m-1$ devrait apparaitre avec une fréquence proche de $1/m$. Le test consiste donc à tirer N valeurs, N étant “grand” ; à compter, pour chaque $i = 0,\dots,m-1$ le nombre N_i de fois que l'on a tiré i ; puis à vérifier qu'aucun des nombres N_i/N n'est “trop différent” de $1/m$. Par exemple, on peut calculer le maximum des valeurs absolues $|N_i/N - 1/m|$ et déclarer l'échec du test si ce maximum est supérieur à une certaine valeur limite α fixée d'avance. Au lieu de calculer le maximum des $|N_i/N - 1/m|$, on peut également calculer leur somme. (Le test du chi-deux, décrit plus loin, propose une approche plus rigoureuse.)

Un autre test plus subtil repose sur l'idée que chaque terme de la suite devrait sembler indépendant du précédent. Pour le vérifier, on tirerait $2N$ fois, d'où une suite $x_0,\dots,x_{2N-1}$; puis on examinerait chacun des N couples (x_{2k},x_{2k+1}), pour $k = 0,\dots,N-1$: et l'on compterait pour tout couple $(i, j) \in \{0,\dots,m-1\}^2$ le nombre $N_{i,j}$ de fois qu'il apparait parmi les (x_{2k},x_{2k+1}). Ce nombre ne devrait pas être “trop différent” de N/m^2.

Enfin, on peut généraliser et tester la fréquence des r-uples : on tire rN fois, on examine les r-uples successifs $(x_{rk},x_{rk+1},\dots,x_{r(k+1)-1})$, on compte le nombre $N_{i_1,\dots,i_r}$ de fois qu'apparait chacun des r-uples possibles $(i_1,\dots,i_r) \in \{0,\dots,m-1\}^r$ et l'on vérifie que, dans l'ensemble, les $N_{i_1,\dots,i_r}$ ne sont pas “trop différents” de rN/m^r.

Exercice 2.2.7 Serait-il intéressant de calculer la somme des $(N_i/N - 1/m)$ (sans valeur absolue) ?

TP 2.2.8 Programmer la mise en oeuvre de ces tests.

Le test du chi-deux

Il s'agit d'une manière scientifiquement fondée de quantifier le jugement “les fréquences ne s'écartent pas trop de la normale”. Que le lecteur se rassure, nous n'en exposerons pas la justification théorique ! De plus, nous n'en décrirons le fonctionnement que dans le cas du test de fréquence simple : les N_i/N s'écartent-ils de $1/m$? On calcule la *variance* :

$$V := mN \sum_{i=0}^{m-1} (N_i/N - 1/m)^2.$$

Puis on utilise la table ci-dessous (voir en haut de la page suivante). L'entier ν désigne le nombre de “degrés de liberté”, dans notre cas, $m-1$; en effet, les N_i/N ne sont pas indépendants les uns des autres : leur somme est nécessairement 1. Supposons que $m = 16$, *i.e.* que l'on ait tiré des nombres entre 0 et 15 : donc $\nu = 15$. D'après la table, on a $V \leq 11,04$ avec une probabilité voisine de 25%. Pour que cette règle soit valable, il faut que le nombre de tirages soit assez grand. Dans notre cas, une règle pratique est $N > 5m$.

TP 2.2.9 Programmer la mise en oeuvre de ce test.

	$p=1\%$	$p=5\%$	$p=25\%$	$p=50\%$	$p=75\%$	$p=95\%$	$p=99\%$
$\nu=1$	0.00016	0.00393	0.1015	0.4549	1.323	3.841	6.635
$\nu=2$	0.02010	0.1026	0.5753	1.386	2.773	5.991	9.210
$\nu=3$	0.1148	0.3518	1.213	2.366	4.108	7.815	11.34
$\nu=4$	0.2971	0.7107	1.923	3.357	5.385	9.488	13.28
$\nu=5$	0.5543	1.1455	2.675	4.351	6.626	11.07	15.09
$\nu=6$	0.8720	1.635	3.455	5.348	7.841	12.59	16.81
$\nu=7$	1.239	2.167	4.255	6.346	9.037	14.07	18.48
$\nu=8$	1.646	2.733	5.071	7.344	10.22	15.51	20.09
$\nu=9$	2.088	3.325	5.899	8.343	11.39	16.92	21.67
$\nu=10$	2.558	3.940	6.737	9.342	12.55	18.31	23.21
$\nu=11$	3.053	4.575	7.584	10.34	13.70	19.68	24.73
$\nu=12$	3.571	5.226	8.438	11.34	14.84	21.03	26.22
$\nu=15$	5.229	7.261	11.04	14.34	18.25	25.00	30.58
$\nu=20$	8.260	10.85	15.45	19.34	23.83	31.41	37.57
$\nu=30$	14.95	18.49	24.48	29.34	34.80	43.77	50.89
$\nu=50$	29.71	34.76	42.94	49.33	56.33	67.50	76.15
$\nu>30$	$\nu+\sqrt{2\nu}x_p+\frac{2}{3}x_p^2-\frac{2}{3}+O(1/\sqrt{\nu})$						
$x_p=$	-2.33	-1.64	$-.675$	0.00	0.675	1.64	2.33

Méthode de Monte-Carlo à l'endroit et à l'envers

La "méthode de monte-Carlo" permet de calculer des intégrales à l'aide de nombres aléatoires. Nous allons l'utiliser à l'envers pour vérifier que des nombres sont aléatoires à l'aide d'intégrales. En fait, ces intégrales n'apparaitront ici qu'en tant qu'aires de parties du plan.

Le principe est le suivant. On choisit m assez grand et l'on tire deux entiers au hasard a,b dans $\{0,\ldots,m-1\}$. Le couple $(x,y) := (a/m, b/m)$ désigne donc un point du carré $K := [0,1]^2$, dont l'aire est 1. Soit maintenant X une partie de K d'aire A. On peut estimer que la probabilité que le point (x,y) soit dans X est A. Si donc on tire N fois de tels couples, le nombre de ceux qui "tombent" dans X ne devrait pas être "trop différent" de NA.

Exercice 2.2.10 On suppose un étang circulaire inscrit dans une parcelle carrée : autrement dit, le cercle et le carré ont même centre et le diamètre du cercle est égal au côté du carré. Des cloches de Pâques [2] larguent en grand nombre des oeufs en chocolat sur toute la campagne environnante. Montrer que, parmi les oeufs tombés dans la parcelle, la proportion de ceux récupérés par les poissons est d'environ $\pi/4$.

TP 2.2.11 Programmer la mise en oeuvre de ce test.

2. Dans la version historique de cet exemple, il s'agissait de tirs de canon.

2.3 Cryptographie

2.3.1 Petit préliminaire théorique

Lemme 2.3.1 *Soient u et v deux entiers premiers entre eux et soit b un entier multiple de u et de v. Alors b est multiple de uv.*

Preuve. - Par hypothèse, il existe des entiers u' et v' tels que $b = uu' = vv'$. D'après le théorème de Bézout, il existe des entiers k et l tels que $ku + lv = 1$. Alors :

$$\begin{aligned} b &= kub + lvb \\ &= kuvv' + lvuu' \\ &= uv(kv' + lu'). \end{aligned}$$

□

Proposition 2.3.2 *Soient p et q deux entiers distincts. Soient $n := pq$ et $n' := (p-1)(q-1)$. Soit enfin $m \geq 2$ un entier tel que :*

$$m \equiv 1 \pmod{n'}.$$

Alors :

$$\forall a \in \mathbf{Z}\ ,\ a^m \equiv a \pmod{n}.$$

Preuve. - Pour montrer que $b := a^m - a$ est multiple de $n = pq$, il suffit, d'après le lemme, de montrer qu'il est multiple de p et multiple de q. Les deux nombres p et q jouant un rôle symétrique, nous ne prouverons que la première assertion. Soit donc $a \in \mathbf{Z}$.

– Si a est multiple de p, alors a^m aussi, et $a^m - a$ également. On a donc bien $a^m \equiv a \pmod{p}$ dans ce cas.
– Si a n'est pas multiple de p, notant $m = 1 + k(p-1)(q-1)$, on déduit du petit théorème de Fermat les congruences :

$$\begin{aligned} a^{p-1} \equiv 1 \pmod{p} &\Longrightarrow a^{k(p-1)(q-1)} \equiv 1 \pmod{p} \\ &\Longrightarrow a^{1+k(p-1)(q-1)} \equiv a \pmod{p} \\ &\Longrightarrow a^m \equiv a \pmod{p}, \end{aligned}$$

qui est la congruence voulue.

□

Remarque 2.3.3 L'entier n' n'est autre que $\phi(n)$ et le théorème d'Euler nous assure que, si a est premier avec n (*i.e.* s'il n'est divisible ni par p ni par q), alors $a^{n'} \equiv 1 \pmod{n}$, d'où l'on déduit sans peine que $a^m \equiv a \pmod{n}$. On pourrait donc prouver la proposition en partant de là : il resterait à examiner les cas où a est divisible par p ou q.

Exercice 2.3.4 Etendre la proposition au cas de r nombres premiers deux à deux distincts.

2.3.2 La méthode RSA

La méthode RSA de cryptographie à clé publique a été inventée en 1978 par Ron Rivest, Adi Shamir et Leonard Adleman du MIT. Sa mise en oeuvre suppose le choix d'un *module* n, d'une *clé publique* (e,n) et d'une *clé secrète* (d,n). Les propriétés requises sont les suivantes :

1. L'entier naturel n est assez grand pour que l'on puisse facilement *coder* (voir la remarque ci-dessous) tout message sous la forme d'une suite d'entiers de $\{0,\dots,n-1\}$.
2. Les entiers naturels d et e vérifient la propriété :

$$\forall a \in \mathbf{Z}\,,\ a^{de} \equiv a \pmod{n}.$$

Remarque 2.3.5 Nous distinguons ici le *codage* du *cryptage*. Le codage est simplement la mise du texte sous une forme adaptée à l'algorithme. Par exemple, chaque caractère du message à transmettre est représenté par un octet et le message est tronçonné en paquets d'octets de tailles telles que l'on puisse les représenter par un entier $M \in \{0,\dots,n-1\}$. Le problème du cryptage est alors, pour l'émetteur du message, de remplacer M par un entier $M' := C(M) \in \{0,\dots,n-1\}$ (la lettre C est ici pour "cryptage"), dont le commun des mortels ne saura que faire, mais que le destinataire du message saura reconvertir en $M := D(M')$ (la lettre D est ici pour "décryptage").

La clé publique (e,n) et la clé secrète (d,n) étant données, le principe est le suivant. Leur détenteur diffuse publiquement (!) la clé publique (e,n) et garde par devers lui (!) la clé secrète (d,n). Chaque fois que quelqu'un veut lui envoyer un message secret $M \in \{0,\dots,n-1\}$:

1. Il crypte M selon la formule $M' := C(M) := M^e \pmod{n}$.
2. Le destinataire décrypte M' selon la formule $M := D(M') := (M')^d \pmod{n}$.

Ici, par abus de notation, $M^e \pmod{n}$ désigne le reste de la division de M^e par n, et similairement pour $(M')^d \pmod{n}$. En fait, ces calculs peuvent s'effectuer relativement vite par exponentiation rapide dans $\mathbf{Z}/n\mathbf{Z}$. La méthode RSA est correcte en vertu du calcul :

$$D(C(M)) = M^{de} \pmod{n} = M.$$

Il reste à voir comment on détermine les clés publique et secrète :

1. On choisit deux grands nombres premiers distincts p et q, et l'on pose $n := pq$.
2. On choisit un entier e premier avec $n' := (p-1)(q-1)$ ni trop grand ni trop petit.
3. On détermine un entier d tel que $de \equiv 1 \pmod{n'}$ à l'aide de l'algorithme d'Euclide étendu.

Cette méthode est praticable parce qu'il est "plutôt facile" de trouver des grands nombres premiers. Si l'on savait aussi facilement factoriser n, la méthode de cryptage serait facile à casser. Mais on ne sait pas factoriser facilement les grands nombres, et, depuis 1978, on n'a pas trouvé non plus d'autres attaques efficaces contre RSA.

TP 2.3.6 Programmer la mise en oeuvre de l'algorithme RSA. On choisira p et q dans une table.

2.4 Suppléments facultatifs

Recherche de la période (méthode de Brent).

Résolution de l'équation $ax \equiv b \pmod{m}$. Résolution simultanée (lemme chinois).

Tests de primalité, factorisation (méthodes élémentaires).

Chapitre 3

Ensembles finis

3.1 Codage par vecteurs de bits

Théorème 3.1.1 *Soit E un ensemble quelconque. A tout sous-ensemble A de E, on associe sa* fonction caractéristique *(également appelée* indicatrice*)* $\chi_A : A \to \{0,1\}$*, qui est définie par :*

$$\forall x \in E \ , \ \chi_A(x) := \begin{cases} 1 \ si \ x \in A, \\ 0 \ si \ x \notin A. \end{cases}$$

Alors l'application $A \mapsto \chi_A$ réalise une bijection de l'ensemble $\mathcal{P}(E)$ des parties de E sur l'ensemble $\{0,1\}^E$ des applications de E dans $\{0,1\}$.

Preuve. - Il suffit de définir la bijection réciproque, c'est-à-dire, pour un élément arbitraire de l'ensemble d'arrivée, de déterminer son unique antécédent. Soit donc $\phi : E \to \{0,1\}$ une application quelconque, autrement dit un élément de $\{0,1\}^E$. Posons $A := \{x \in E \mid \phi(x) = 1\}$. C'est un sous-ensemble de E, et c'est bien l'unique sous-ensemble tel que $\chi_A = \phi$, donc l'unique antécédent de ϕ par l'application $A \mapsto \chi_A$. □

Ce théorème fournit une méthode pour *coder* les sous-ensemble d'un ensemble fini quelconque E. Pour cela, notant $n :=$ card E le nombre d'éléments de E, on commence par numéroter ces éléments de 0 à $n-1$. Cela revient à *identifier* E avec l'ensemble particulier $E_n := \{0,\dots,n-1\}$. Dorénavant, nous travaillerons directement sur l'ensemble E_n.

Une application $\phi : E_n \to \{0,1\}$ est alors totalement spécifiée par le n-uplet $(x_0,\dots,x_{n-1}) \in \{0,1\}^n$ de ses valeurs $x_i := \phi(i)$, $i = 0,\dots,n-1$. Autrement dit, une partie A de E_n est représentée par un *vecteur de bits* ("bit-vector") $X := (x_0,\dots,x_{n-1}) \in \{0,1\}^n$, avec les règles :

$$x_i = 1 \Longleftrightarrow i \in A,$$
$$x_i = 0 \Longleftrightarrow i \notin A.$$

Par exemple, l'ensemble vide $\emptyset$ est codé par le vecteur nul $(0,\dots,0) \in \{0,1\}^n$ et l'ensemble "plein" E_n est codé par le vecteur $(1,\dots,1) \in \{0,1\}^n$.

TP 3.1.2 Réaliser ce codage et programmer la fonction associée de test d'appartenance.

Exercice 3.1.3 Montrer que l'on peut coder les parties finies de $\mathbf{N}$ à l'aide de la bijection $A \mapsto \sum_{n \in A} 2^n$ de l'ensemble $\mathcal{P}_f(\mathbf{N})$ des parties finies de $\mathbf{N}$ sur l'ensemble $\mathbf{N}$.

3.2 Calcul booléen sur les sous-ensembles

Le codage décrit ci-dessus permet d'arithmétiser le calcul sur les sous-ensembles de E_n. Ainsi, si $X := (x_0, \ldots, x_{n-1}) \in \{0,1\}^n$ est le vecteur de bits associé au sous-ensemble A de E_n, le vecteur de bits associé au complémentaire $\complement A$ de A dans E_n est :

$$\overline{X} := (\overline{x_0}, \ldots, \overline{x_{n-1}}) \in \{0,1\}^n,$$

où l'on a posé $\overline{0} := 1$ et $\overline{1} := 0$.

Soient maintenant $X := (x_0, \ldots, x_{n-1}) \in \{0,1\}^n$ et $Y := (y_0, \ldots, y_{n-1}) \in \{0,1\}^n$ les vecteurs de bits respectivement associés à deux sous-ensembles A et B de E_n. Alors :

$$i \in A \cap B \Longleftrightarrow i \in A \text{ et } i \in B \Longleftrightarrow x_i = 1 \text{ et } y_i = 1 \Longleftrightarrow x_i y_i = 1,$$

de sorte que le vecteur de bits associé à $A \cap B$ est le vecteur :

$X.Y := (x_0.y_0, \ldots, x_{n-1}.y_{n-1})$, avec la loi de composition décrite par la table :

.	0	1
0	0	0
1	0	1

De même :

$$i \in A \cup B \Longleftrightarrow i \in A \text{ ou } i \in B \Longleftrightarrow x_i = 1 \text{ ou } y_i = 1 \Longleftrightarrow x_i + y_i = 1,$$

de sorte que le vecteur de bits associé à $A \cup B$ est le vecteur :

$X.Y := (x_0.y_0, \ldots, x_{n-1}.y_{n-1})$, avec la loi de composition décrite par la table :

+	0	1
0	0	1
1	1	1

Attention ! Il ne s'agit pas de l'addition de $\mathbf{Z}/2\mathbf{Z}$, que nous notons ici $\oplus$ et qui est rappelée dans table ci-dessous :

$\oplus$	0	1
0	0	1
1	1	0

Cette opération code la *différence symétrique* :

$$A \vartriangle B := (A \cap \complement B) \cup (B \cap \complement A) = (A \setminus B) \cup (B \setminus A).$$

On a noté $A \setminus B := A \cap \complement B$ la différence de deux ensembles. Par exemple $\complement A = E_n \vartriangle A$ et $\overline{X} = (1, \ldots, 1) \oplus X$.

Enfin, la relation d'inclusion $A \subset B$ entre deux sous-ensembles de E_n peut également s'exprimer en termes des vecteurs de bits X et Y associés. Elle équivaut en effet à $A \cap B = A$, donc à la relation $X.Y = X$. Elle équivaut d'ailleurs également à $A \cup B = B$, donc à la relation $X + Y = Y$.

TP 3.2.1 Programmer ces fonctions.

3.3 Le produit cartésien

Le produit cartésien $E \times F$ des ensembles E et F est, par définition, l'ensemble des couples (x,y) avec $x \in E$ et $y \in F$. On a $\text{card}(E \times F) = (\text{card}E)(\text{card}F)$. Si l'on a identifié E à E_n et F à E_p, leur produit cartésien a donc np éléments et l'on est conduit à *rechercher une bijection entre* $E_n \times E_p$ *et* E_{np}. Cela revient donc à *ordonner les éléments de* $E_n \times E_p$.

Prenons pour fixer les idées $n := 3$ et $p := 4$. Les 12 éléments de $E_3 \times E_4$ sont naturellement rangés en un tableau rectangulaire (une "matrice") :

$$\begin{matrix} (0,0) & (0,1) & (0,2) & (0,3) \\ (1,0) & (1,1) & (1,2) & (1,3) \\ (2,0) & (2,1) & (2,2) & (2,3) \end{matrix}$$

Il y a deux parcours naturels de cette matrice : par lignes ou par colonnes. Le premier fournit l'ordre suivant :

$$(0,0),(0,1),(0,2),(0,3),(1,0),(1,1),(1,2),(1,3),(2,0),(2,1),(2,2),(2,3).$$

Si l'on veut numéroter ces couples de 0 à 11, on voit que le couple (i,j) reçoit le numéro $4i + j$. Le second parcours fournit l'ordre suivant :

$$(0,0),(1,0),(2,0),(0,1),(1,1),(2,1),(0,2),(1,2),(2,2),(0,3),(1,3),(2,3).$$

Si l'on veut numéroter ces couples de 0 à 11, on voit que le couple (i,j) reçoit le numéro $i + 3j$.

Théorème 3.3.1 *(i) L'application $(i,j) \mapsto pi + j$ réalise une bijection du produit cartésien $E_n \times E_p$ sur E_{np}.*
(ii) L'application $(i,j) \mapsto i + nj$ réalise une bijection du produit cartésien $E_n \times E_p$ sur E_{np}.

Preuve. - Dans les deux cas, il suffit d'exhiber l'application réciproque. Dans le premier cas, c'est $k \mapsto (i,j)$ ou i est le quotient et j le reste de la division euclidienne de k par p ; dans le second cas, c'est $k \mapsto (i,j)$ ou j est le quotient et i le reste de la division euclidienne de k par n. □

TP 3.3.2 Programmer ce codage

Exercice 3.3.3 L'ensemble des parties de E_n a 2^n éléments, il est donc en bijection avec E_{2^n}. Expliciter un algorithme qui permette de convertir un élément de $E_{2^n} = \{0, 1, \ldots, 2^n - 1\}$ en un vecteur de n bits.

Chapitre 4

Recherche et tri

4.1 Algorithmes de recherche

Nous allons examiner trois problèmes naturels de recherche dans un tableau dont les solutions illustrent :

- différentes classes de coûts des algorithmes ;
- l'influence des structures de données utilisées ;
- l'intérêt de certaines méthodes *dichotomiques*, associées au paradigme "diviser pour régner".

De plus, les problèmes de recherche motivent l'étude des algorithmes de tri, qui seront évoqués à la section suivante.

4.1.1 Recherche d'un élément

On se donne un tableau $T[1],\ldots,T[n]$ d'éléments d'un type arbitraire non spécifié. L'exemple canonique est celui d'un tableau de chaines de caractères. On cherche un élément particulier e dans le tableau T : autrement dit, s'il en existe, un indice $i \in \{1,\ldots,n\}$ tel que $T[i] = e$.

Cas d'un tableau arbitraire

L'algorithme de base est alors le suivant (le tableau est noté `tab`) :

```
i := 1;
tant que (tab[i] <> e et i <= n) faire
   i := i+1;
fin tant que;
si (i <= n) alors
   rendre (i)
sinon
   ECHEC;
fin si;
```

Pour estimer le coût de cet algorithme, nous prendrons pour critère le nombre de comparaisons effectuées (de $T[i]$ avec e). Si l'élément e recherché n'est pas dans le tableau, ce nombre est n. Si l'élément e recherché est dans le tableau, ce nombre est compris entre 1 et n. Le meilleur des cas

est celui où $T[1] = e$, qui donne un coût de 1. Le pire des cas est celui où $T[n] = e$ et $T[i] \neq e$ pour $i < n$, qui donne un coût de n.

Et en moyenne, quel est le coût ? Si nous supposons que l'élément e peut être trouvé dans chacune des positions $T[1], \dots, T[n]$ avec la même probabilité, le coût moyen est :

$$\frac{1+\cdots+n}{n} = \frac{n(n+1)/2}{n} = \frac{n+1}{2}.$$

L'hypothèse d'équiprobabilité des positions signifie concrètement que, lorsque l'on effectue un très grand nombre N de recherches d'éléments qui figurent effectivement dans le tableau, on trouve la position $T[1]$ approximativement N/n fois, et de même pour $T[2], \dots, T[n]$. Cette hypothèse est raisonnable si tous les éléments du tableau sont distincts. Si ce n'est pas le cas, l'analyse est légèrement plus compliquée, mais le résultat n'est pas substantiellement différent.

En conclusion, le coût d'une recherche avec échec est constant égal à n ; le coût d'une recherche avec succès est au pire égal à n, et en moyenne égal à $\frac{n+1}{2}$: le coût est donc *linéaire*.

Remarque 4.1.1 Nous éludons complètement la question des *biais d'utilisation du tableau* : par exemple les phénomènes socioculturels qui font qu'un utilisateur cherchera plus fréquemment la chaine de caractères "Michael Jackson" que la chaine de caractères "Coût des algorithmes".

Cas d'un tableau trié

On suppose maintenant que le type des éléments du tableau est muni d'une relation d'ordre : l'ordre évident s'il s'agit de nombres, et, par exemple, l'ordre lexicographique (c'est-à-dire l'ordre du dictionnaire) s'il s'agit de chaines de caractères. Supposons de plus que le tableau est *trié*, autrement dit, que l'on ait :

$$T[1] \leq \cdots \leq T[n].$$

L'algorithme ci-dessus peut alors facilement être modifié pour détecter un échec avant la fin de la lecture complète du tableau. En effet, dès que $T[i] \geq e$, on sait que l'on n'a pas besoin de chercher plus loin. Voici la version améliorée (le tableau est encore noté `tab`) :

```
i := 1;
tant que (tab[i] < e et i <= n) faire
   i := i+1;
fin tant que;
si (i <= n et tab[i] = e) alors
   rendre (i);
sinon
   ECHEC;
fin si;
```

Il est facile de vérifier que le coût d'une recherche avec succès n'est pas modifié par cette amélioration. Quant au coût d'une recherche avec échec, on peut estimer (en l'absence d'hypothèse particulière sur les valeurs effectivement recherchées par les utilisateurs) qu'il est en moyenne divisé par deux. C'est une amélioration, mais mineure.

On peut faire beaucoup mieux pour exploiter le fait que le tableau est trié. Une recherche *dichotomique* consiste à regarder au milieu du tableau ; puis, si l'on n'a pas trouvé, à aller à gauche ou à droite selon le cas. La taille du domaine de recherche est divisée par deux à chaque étape, ce qui garantit une terminaison beaucoup plus rapide. Voici une façon d'écrire l'algorithme :

```
i := 1;
j := n;
trouvé := faux;
tant que (non trouvé et i <= j) faire
   k := (i+j) div 2;
   si (tab[k] = e) alors
      trouvé := vrai;
   sinon
      si (tab[k] < e) alors
         i := k+1;
      sinon
         j := k-1;
      fin si;
   fin si;
fin tant que;
si trouvé alors
   rendre (i);
sinon
   ECHEC;
fin si;
```

(On a fait appel à la fonction `floor` qui calcule la partie entière d'un réel.) A chaque étape, la longueur $j-i+1$ de l'intervalle de recherche est au moins divisée par deux, puisque sa borne gauche passe à droite de son centre ou sa borne droite à gauche de son centre. Le nombre total d'étapes est donc au pire $\log_2 n$. Le coût le pire d'une recherche avec succès ou avec échec est donc majoré par $\log_2 n$. Nous ne calculerons pas le coût moyen, mais on peut démontrer qu'il est proche du coût le pire.

Quelques conclusions

Nous allons d'abord discuter d l'interêt davoir un coût logarithmique plutôt que linéaire. A une époque ou la puissance des matériels croît de manière si spectaculaire, cela vaut-il la peine de se soucier de la qualité des algorithmes ? Pour le voir, nous prendrons comme hypothèse heuristique la “loi de Moore”, une loi socio-économico-technique qui dit que “la puissance des matériels informatiques est multipliée par deux tous les dix-huit mois”.

Remarque 4.1.2 La “loi de Moore” n'a rien d'une loi scientifique, c'est, au mieux, une constatation empirique ; voir une discussion là-dessus dans l'article de Wikipedia consultable sur l'URL `http://fr.wikipedia.org/wiki/Loi_de_Moore`.

On se place dans la situation suivante : nous avons un ordinateur et un programme qui nous permettent de retrouver un nom dans une liste de cinq cents éléments en un dixième de secondes. Il est donc bien adapté à la gestion de l'ensemble des étudiants en informatique de l'UPS, par

exemple. Pouvons-nous utiliser le même ordinateur et le même programme dans le cas d'une liste de cinq cents mille éléments (population de la ville de Toulouse, par exemple) ? Combien de temps faudra-t-il alors pour trouver un élément dans cette liste ?

Avec notre premier algorithme (coût linéaire), le temps d'exécution est proportionnel à la taille de la liste (du tableau). Il faut donc mille fois plus de temps, soit cent secondes : ce n'est pas viable. Cette dégradation peut-elle être compensée par les progrès de la technologie ? Quand ces progrès nous permettront-ils de travailler sur une telle grosse liste mais avec le temps de réponse actuel ? Nous interpréterons la loi de Moore comme nous disant que les machines vont deux fois plus vite tous les dix-huit mois, donc $2^{10} = 1024$ fois plus vite au bout de 180 mois. Il nous faudra donc attendre quinze ans pour pouvoir utiliser notre algorithme à coût linéaire à l'échelle de Toulouse avec le même temps de réponse qu'auparavant.

Avec notre second algorithme (coût logarithmique), le temps d'exécution est de la forme $C\log_2 n$, avec une constante C telle que $C\log_2 500 = 1/10$ (secondes). A l'échelle de Toulouse, cela donne un temps d'exécution de :

$$C\log_2 500000 = 1/10 \times (\log_2 500000/\log_2 500) \simeq 1/10 \times 2,2 \text{ secondes.}$$

Le temps d'exécution n'est pas outrageusement plus long, et il nous faudra seulement attendre un peu plus de dix-huit mois pour pouvoir utiliser notre algorithme à coût logarithmique à l'échelle de Toulouse avec le même temps de réponse qu'auparavant :

Les bons algorithmes profitent mieux de l'amélioration du matériel.

Reste que notre bon algorithme suppose le tableau trié. Le tri d'un tableau est lui-même une opération couteuse ! Pratiquement, il n'est pas question de trier un tableau dans lequel on n'effectuera qu'une ou deux recherches ; mais, si les recherches doivent être fréquentes (dictionnaire, base de donnéees ...), autant trier le tableau (ou la liste, ou le fichier selon le cas) [1].

Exercice 4.1.3 On supose que le tableau est non trié et contient des entiers compris entre 1 et m. Il y a donc m^n possibilités de tels tableaux, que l'on considèrera comme équiprobables. Combien de comparaisons sont en moyenne nécessaires pour trouver un élément $e \in \{1,\dots,m\}$?

TP 4.1.4 Programmer l'algorithme de recherche dichotomique.

4.1.2 Recherche du maximum dans un tableau

On suppose encore que le type des éléments du tableau est muni d'une relation d'ordre, et l'on se propose de rechercher la position du plus grand des éléments $T[1],\dots,T[n]$. Bien entendu, si le tableau est trié, ce maximum est $T[n]$ et le problème est trivial. En général, voici un algorithme simple :

1. On admet implicitement ici que le tableau est constitué et trié une fois pour toutes, ce qui est bien le cas pour un dictionnaire, mais rarement pour une base de données. Pour un ensemble de données qui évolue, d'autres structures de données conviennent.

```
i := 1;
j := 1;
m := tab[1];
tant que (i < n) faire
   i := i+1;
   si (m < tab[i]) alors
      m := tab[i];
      j := i;
   fin si;
fin tant que;
rendre(j,m);
```

A toute étape de la boucle, $m = T[j]$ est le maximum de $T[1],\ldots,T[i]$. Le nombre de comparaisons est toujours $n-1$. Comme critère de coût, nous prendrons plutôt le nombre de fois où l'on a dû mettre à jour j et m (initialisations comprises).

Dans le meilleur des cas, le maximum se trouve en $j = 1$ et le coût est donc de 1 (initialisations). Dans le pire des cas, le tableau est trié (mais on ne le sait pas !) et j prend successivement toutes les valeurs de 1 à n : le coût est donc de n. En général, le coût est égal au nombre de "maxima stricts gauche-droite", c'est-à-dire au nombre des indices i tels que $T[i] > \max(T[1],\ldots,T[i-1])$. On peut démontrer que, si le tableau contient n éléments distincts, et que les différents ordres possibles de ces éléments sont équiprobables, alors le coût moyen est égal à $H_n := 1 + \frac{1}{2} + \cdots + \frac{1}{n}$ (ces nombres forment la *série harmonique*).

Exemple 4.1.5 Si $n = 1$, le seul rangement possible (!) donne un coût de $1 = H_1$. Si $n = 2$, les deux rangements possibles donnent respectivement des coûts de 1 et 2, d'où un coût moyen de $3/2 = H_2$. Pour le cas $n = 3$, on peut aussi bien supposer que les éléments du tableau sont 1, 2 et 3 pris dans un certain ordre. Il y a six permutations possibles : (123), (132), (213), (231), (312) et (321). Les coûts correspondants sont respectivement 3, 2, 2, 2, 1 et 1, d'où un coût moyen de $11/6 = H_3$.

On démontrera en cours d'analyse de L1 que $H_n \sim \ln n$, ce qui implique que le coût de l'algorithme ci-dessus est logarithmique.

Exercice 4.1.6 En comparant $\int_i^{i+1} \frac{dt}{t}$ à $\int_i^{i+1} \frac{dt}{i}$ et à $\int_i^{i+1} \frac{dt}{i+1}$, démontrer l'encadrement :

$$\frac{1}{i+1} \leq \ln(i+1) - \ln(i) \leq \frac{1}{i}.$$

En déduire par récurrence l'encadrement $\ln(n+1) \leq H_n \leq 1 + \ln n$, puis l'équivalence $H_n \sim \ln n$.

4.1.3 Recherche d'un duplicata

Pour commencer, on ne fait aucune hypothèse sur le type des éléments du tableau. Le problème est de déterminer, s'il en existe, des indices distincts $i \neq j$ tels que $T[i] = T[j]$. La seule méthode envisageable est *a priori* d'effectuer toutes les comparaisons de couples $T[i], T[j]$ pour $1 \leq i < j \leq n$, ce qui représente $n(n-1)/2$ comparaisons. Ce nombre correspond au cas le pire (pas de

duplicata, il faut tout avoir essayé avant de s'en apercevoir). En moyenne, en cas de succès, on peut s'attendre à un coût moitié moindre, donc de l'ordre de $n^2/2$. Il s'agit donc d'un coût *quadratique*.

Exemple 4.1.7 Reprenant nos considérations heuristiques basées sur la "loi de Moore". Un algorithme quadratique qui s'exécute en un dixième de seconde sur un tableau de cinq cents éléments prendra un million de fois plus de temps pour un tableau de cinq cents mille éléments. Il faudra $20 \times 18 = 360$ mois, soit trente ans, pour que l'amélioration du matériel informatique permette de compenser cette dégradation de performance.

Si l'on suppose maintenant que le type des éléments du tableau est muni d'une relation d'ordre et que le tableau est trié. Pour déceler d'éventuelles duplications, il suffit de considérer des paires d'éléments consécutifs du tableau. (Voyez-vous pourquoi ?) On a donc besoin d'au plus $(n-1)$ comparaisons. Le coût est devenu linéaire, ce qui est beaucoup mieux. Nous verrons que l'on peut trier un tableau de taille n avec un coût de l'ordre de $n \ln n$: donc, même pour chercher les duplications une seule fois, cela vaut la peine de trier préalablement le tableau. (Bien entendu, cette affirmation ne concerne que des tableaux assez grands pour que le coût soit un problème !)

TP 4.1.8 Programmer la recherche de duplicata dans un tableau non trié et vérifier l'estimation du coût moyen par expérimentation sur des tableaux créés à l'aide d'un générateur aléatoire.

4.2 Algorithmes de tri

Il y a d'innombrables bonnes raisons pour trier des tableaux, des listes, des fichiers ..., l'une d'entre elles étant que cela donne lieu à des algorithmes intéressants et dont l'analyse est elle-même intéressante !

Dans ce qui suit, on suppose tous les éléments considérés sont de même type et que ce type est muni d'une relation d'ordre.

4.2.1 Tri de trois éléments

Commençons par un cas d'apparence anodin. Soient a,b,c trois éléments distincts. On veut produire un triplet (a',b',c') tel que :

1. le triplet (a',b',c') soit une permutation du triplet (a,b,c) ;
2. on ait $a' < b' < c'$.

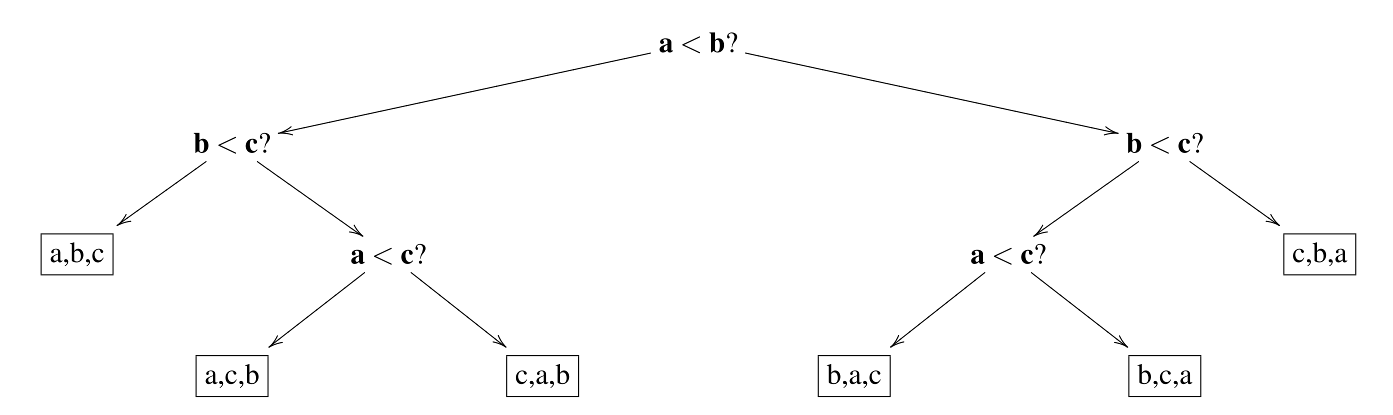
Pour cela, on s'autorise des tests qui sont des comparaisons : $\mathbf{a} < \mathbf{b}$?, $\mathbf{a} < \mathbf{c}$? et $\mathbf{b} < \mathbf{c}$?. Il peut arriver que deux tests consécutifs soient suffisants pour trancher : si $a < b$ et $b < c$ sont tous deux vrais, alors $(a',b',c') = (a,b,c)$. De même si $a < b$ et $b < c$ sont tous deux faux, alors $(a',b',c') = (c,b,a)$. Mais si $a < b$ est vrai et que $b < c$ est faux, il y a deux possibilités : $(a',b',c') = (a,c,b)$ ou $(a',b',c') = (c,a,b)$, et il faudra un troisième test $a < c$? pour trancher. Dans tous les cas, on devrait s'en tirer avec un nombre de comparaisons compris entre deux et trois. Pour aller plus loin, écrivons un algorihme possible [2] :

2. Si l'on veut trier un nombre indéterminé n d'éléments, il ne sera évidemment pas possible d'écrire un algorithme basé sur des "si ...alors ...sinon ..." comme celui-ci !

```
si (a < b) alors
   si (b < c) alors
      rendre (a,b,c);
   sinon
      si (a < c) alors
         rendre (a,c,b);
      sinon
         rendre (c,a,b);
      fin si;
   fin si;
sinon
   si (b < c) alors
       si (a < c) alors
          rendre (b,a,c);
       sinon
          rendre (b,c,a);
       fin si;
   sinon
      rendre (c,b,a);
   fin si;
fin si;
```

Si l'on suppose les six ordres $a < b < c$, $a < c < b$, $b < a < c$, $b < c < a$, $c < a < b$ et $c < b < a$ équiprobables, on voit que le nombre moyen de comparaisons effectuées (donc le coût moyen) est $\dfrac{2+3+3+3+3+2}{6} = 8/3 \simeq 2,67$.

La procédé des tests successifs avec réponse binaire oui/non peut être modélisé par un *arbre de décision*, ici :



Les **noeuds** de l'arbre représentent des tests, la branche gauche (resp. droite) correspondant à une réponse positive (resp. négative) ; les feuilles représentent les résultats possibles de l'algorithme.

La distance d'une feuille à la *racine* (qui est le noeud le plus haut en informatique, donc ici $\mathbf{a} < \mathbf{b}$? ! ! !) est appelée *profondeur* de cette feuille, et elle est dans notre cas égale au coût de l'exécution qui a amené à cette feuille. On voit bien sur cet arbre pourquoi on n'aurait pas pu s'en

tirer avec deux comparaisons dans tous les cas. Cela signifierait en effet que toutes les feuilles auraient une profondeur inférieure ou égale à 2, et l'arbre ne pourrait alors avoir que 4 feuilles. Mais cela ne suffirait pas, puisque l'on sait d'avance qu'il y a 6 ordres possibles, donc qu'il faut au moins 6 feuilles.

Coût optimal d'un tri par comparaisons

Ce raisonnement se généralise ainsi. Supposons que l'on veuille trier n éléments en effectuant [3] un certain nombre de comparaisons entre eux. L'arbre de décision correspondant doit avoir au moins $n!$ feuilles [4] puisque l'on doit prévoir au moins $n!$ résultats différents possibles. Par ailleurs, si l'on veut effectuer au plus k comparaisons dans tous les cas, les feuilles de l'arbre ont toutes une profondeur inférieure ou égale à k, et l'on voit facilement (avec de petits dessins de tels arbres) qu'il y a alors au plus 2^k feuilles. En résumé, si l'on veut trier n éléments en effectuant dans tous les cas au plus k raisons, on doit avoir :

$$2^k \geq n! \Longrightarrow k \geq \log_2 n!.$$

Théorème 4.2.1 *Le nombre de comparaisons nécessaires pour trier n éléments est, dans le pire des cas, au moins égal à $\log_2 n!$. On a de plus :*

$$\log_2 n! \sim n \log_2 n.$$

(La deuxième assertion est justifiée par l'exercice ci-dessous.) On peut démontrer une estimation analogue pour le coût moyen d'un tel algorithme. Comme nous le verrons plus loin, ces ordres de grandeur en $n \log_2 n$ peuvent effectivement être réalisés par de bons algorithmes.

Exercice 4.2.2 Démontrer l'encadrement :

$$1 + n \ln n \leq (n+1)\ln(n+1) - n \ln n \leq 1 + \ln(n+1).$$

En déduire par écurrence un encadrement de $\ln 1 + \ln 2 + \cdots + \ln n = \ln n!$, puis une preuve de la deuxième assertion du théorème.

4.2.2 Tri par sélection

On considère un tableau $T[1], \ldots, T[n]$. Le principe de l'algorithme qui va suivre est le suivant : on détermine le plus grand élément $T[j]$ du tableau, et l'on permute $T[j]$ avec $T[n]$, qui est alors en place définitivement ; puis on détermine le plus grand élément du sous-tableau $T[1], \ldots, T[n-1]$, que l'on permute avec $T[n-1]$, etc. L'invariant de boucle qui permet de démontrer la correction de l'algorithme est donc le suivant : après k étapes, les k derniers éléments du tableau sont en place ; autrement dit, ce sont les k plus grands et ils sont triés. (Il s'agit évidemment de la boucle *externe*, la boucle *interne* visant chaque fois à déterminer le maximum des $(n-k)$ premiers éléments.)

3. Il existe des méthodes de tri radicalement différentes qui ne reposent pas sur une séquence de comparaisons ; ni le raisonnement qui suit, ni sa conclusion ne s'appliquent à ces méthodes.

4. On rappelle que $n!$ désigne la factorielle de n, c'est-à-dire le produit $1 \times 2 \times \cdots \times n$ des n premiers entiers non nuls ; et que le nombre de permutations de n éléments distincts est égal à $n!$.

```
i := n;
tant que (i > 1) faire
   tab[j] := le maximum de tab[1]...tab[i];
   permuter tab[i] et tab[j];
   i := i - 1
fin tant que;
```

Noter que l'on n'a rien à faire pour $i = 1$: lorsque les $(n-1)$ derniers éléments sont en place, le premier l'est également. La recherche du maximum de $T[1],\ldots,T[i]$ coûte $(i-1)$ comparaisons, le nombre total de comparaisons effectué est donc dans tous les cas égal à $(n-1)+(n-2)+\cdots+1 = n(n-1)/2$; c'est donc un coût quadratique. On peut cependant choisir d'autres critères. Par exemple le nombre total de mises à jour de l'indice lors des recherches de maximum, mais ce nombre est difficile à analyser.

Un autre critère pertinent est le nombre d'échanges de $T[i]$ avec $T[j]$. En effet, si les éléments du tableau sont volumineux, ces échanges peuvent prendre un temps non négligeable. Dans notre algorithme, il y a dans tous les cas $(n-1)$ échanges, même si par exemple le tableau est déjà trié et que l'on a chaque fois $i = j$! On peut améliorer l'algorithme sur ce point et n'effectuer l'échange que s'il est nécessaire :

```
i := n;
tant que (i > 1) faire
   tab[j] := le maximum de tab[1]...tab[i];
   si (i <> j) alors
      permuter tab[i] et tab[j];
   fin si;
   i := i - 1;
fin tant que;
```

Le nombre minimum d'échanges est alors 0 (tableau déjà trié). Le nombre maximum est $n-1$; il ne correspond d'ailleurs pas à un tableau en ordre inverse $n,\ldots,1$ mais, par exemple, à l'ordre $2,3,\ldots,n,1$. On peut prouver que le nombre moyen d'échanges est $n-H_n$. L'amélioration n'apporte donc pas vraiment grand-chose, puisque $H_n \sim \ln n$ est négligeable devant n.

Exercice 4.2.3 Combien y a-t-il d'échanges dans le cas de l'ordre inverse $n,\ldots,1$?

TP 4.2.4 Programmer la version améliorée de l'algorithme.

4.2.3 Tri par fusion

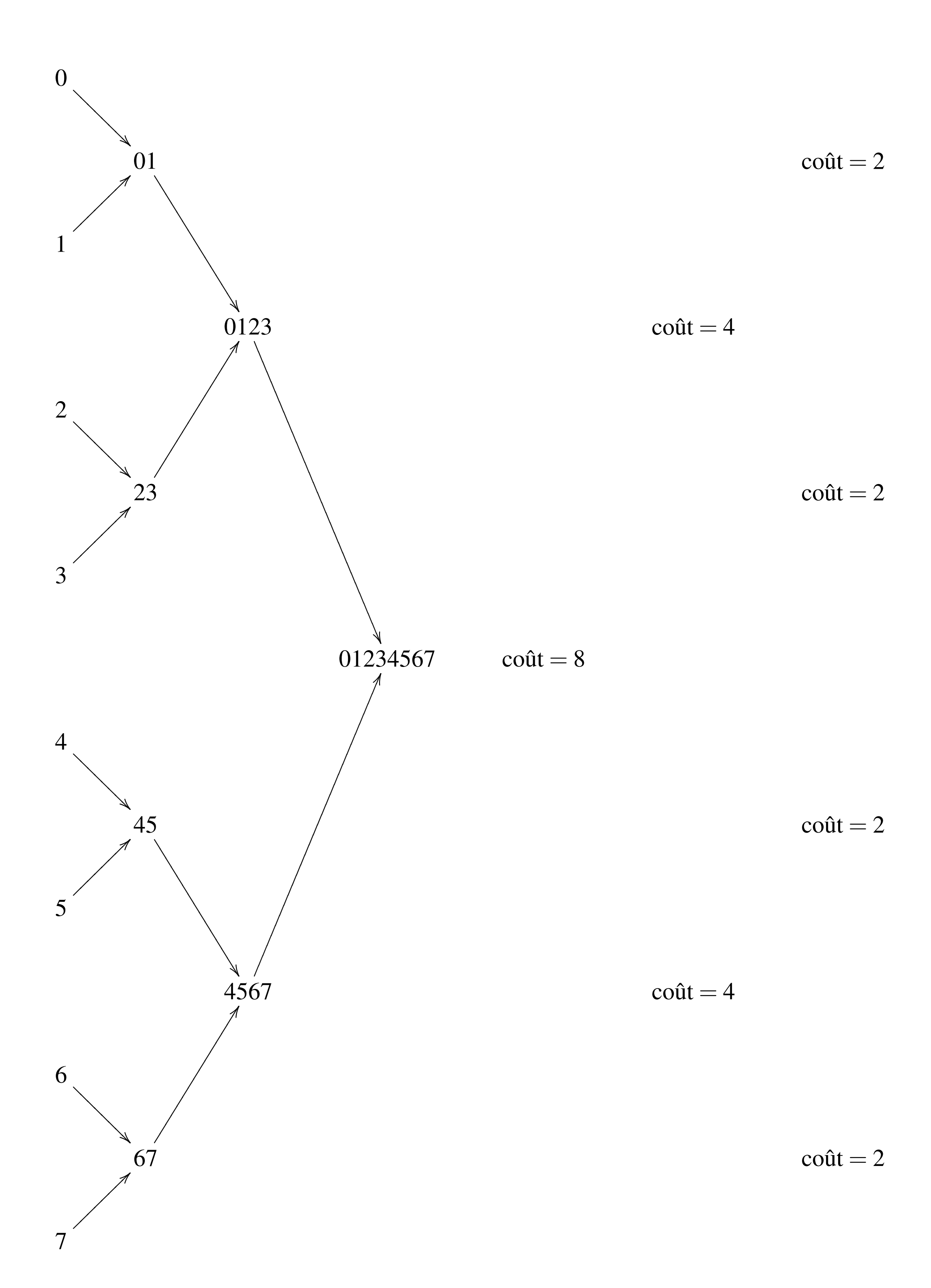
Nous allons décrire un algorithme de tri *dichotomique*. Le nombre de comparaisons effectuées par cet algorithme dans le cas d'un tableau de n éléments, est de l'ordre de $n\log_2 n$: c'est donc l'ordre de grandeur optimal. Cela ne signifie d'ailleurs pas pour autant que l'algorithme lui-même est optimal ; mais les autres critères pertinents pour en juger (difficulté de mise en oeuvre, coût des opérations élémentaires, adéquation de la structure de données ...) et la comparaison pratique des différents algorithmes en situation relèvent du cours d'informatique.

Description informelle

Le principe est le suivant. Pour trier un tableau de seize éléments (par exemple), on coupe celui-ci en deux sous-tableaux de huit éléments ; on trie chacun de ces sous-tableaux, puis on *fusionne* ces derniers. Le tri des tableaux de huit éléments a été effectué de la même manière : on les a coupés en tableaux de quatre éléments, que l'on a triés puis fusionné, etc. Cette description appelle plusieurs remarques :

1. L'algorithme est "récursif" : son exécution sur un gros tableau *présuppose* son exécution sur des tableaux plus petits. Pour ne pas se retrouver dans un cercle vicieux, il faut donc que le tri de très petits tableaux soit direct et ne fasse pas appel au tri de tableaux encore plus petits. C'est bien entendu le cas de tableaux à un élément, dont le tri est une opération vide ! Pratiquement, nous ne décrirons pas l'algorithme sous sa forme récursive mais sous la forme d'une itération directe : d'abord les tableaux à un élément, puis les tableaux à deux éléments, puis à quatre éléments, etc. (La programmation récursive, c'est pour plus tard !)
2. On doit disposer d'une méthode pour "fusionner" deux tableaux triés de huit éléments en un tableau trié de seize éléments. Nous verrons qu'il existe en effet une telle méthode efficace, qui, de manière générale, fusionne deux tableaux triés ayant respectivement ℓ et m éléments en un tableau trié de $n := \ell + m$ éléments. Nous prouverons que le coût d'une telle fusion est, dans le pire des cas, égal à n : ceci, que l'on prenne comme unités de compte les comparaisons ou les affectations d'éléments du tableau.
3. Pour que des divisions successives par deux des tailles de tableaux nous ramènent à des tableaux de taille un, il faut que la taille de départ soit une puissance de deux (seize dans notre exemple). Si l'on part d'un tableau ayant par exemple douze éléments, il suffira de le compléter par quatre éléments inutiles de valeur maximale à la fin. Ces éléments resteront à leur place et, à la fin, seuls les douze premiers éléments du résultat nous intéresseront. Nous verrons que ce procédé de remplissage (qui est souvent nécessaire dans les algorithmes dichotomiques) n'a pas d'influence sur l'ordre de grandeur du coût.
4. Nous nous restreindrons donc à des tableaux de $n = 2^k$ éléments. Pour simplifier la manipulation des indices des éléments , nous supposerons que ceux-ci varient de 0 à $n-1$ (comme en C, par exemple). En effet, dans ce cas, les indices des deux sous-tableaux varient respectivement de 0 à $2^{k-1}-1$ et de 2^{k-1} à 2^k-1 : les formules sont plus simples qu'elles ne le seraient pour des indices variant de 1 à n.

Dans la figure suivante, on détaille le cas d'un tableau de huit éléments. Par exemple, en fusionnant le sous-tableau réduit à l'élément $T[0]$ avec le sous-tableau réduit à l'élément $T[1]$, on obtient le sous-tableau trié formé des éléments $T[0], T[1]$ et le coût de cette fusion est 2 (pour le moment, on admet qu'il existe une telle méthode de fusion). De même, en fusionnant le sous-tableau trié formé des éléments $T[0], T[1]$ avec le sous-tableau trié formé des éléments $T[2], T[3]$, on obtient le sous-tableau trié formé des éléments $T[0], T[1], T[2], T[3]$ et le coût de cette fusion est 4. Le total des coûts est donc : $4 \times 2 + 2 \times 4 + 1 \times 8 = 24$.



Coût

Le calcul se généralise ainsi au cas d'un tableau de 2^k éléments. Les fusions deux à deux des 2^k sous-tableaux de 1 éléments nous donnent 2^{k-1} sous-tableaux triés de 2 éléments, pour un

coût total de $2^{k-1} \times 2 = 2^k$. Les fusions deux à deux des 2^{k-1} sous-tableaux triés de 2 éléments nous donnent 2^{k-2} sous-tableaux triés de 4 éléments, pour un coût total de $2^{k-2} \times 4 = 2^k$; et ainsi de suite. Chaque étape fusionnant 2^{k-i} sous-tableaux triés de 2^i éléments a pour coût total $2^{k-i} \times 2^i = 2^k$, et il y a k telles étapes (pour i variant de 0 à $k-1$) : le coût total est donc $k2^k$. Comme $n = 2^k$, on a bien un coût égal à $n \log_2 n$.

Remarque 4.2.5 Si l'on a eu à trier pour commencer un nombre n d'éléments qui n'est pas une puissance de 2, après remplissage, on a travaillé sur un tableau de 2^k éléments avec $2^{k-1} < n \leq 2^k$, donc $k-1 < \log_2 n \leq k$, et le coût total de $k2^k$ est compris entre $n \log_2 n$ et $2n(1+\log_2 n)$: l'ordre de grandeur est bien le même.

On dit d'un tel algorithme que son coût est *quasi-linéaire* ; voici pourquoi. Les différentes classes de coût que nous avons rencontrées (logarithmique, linéaire, quadratique) peuvent être caractérisées par la manière dont le coût augmente lorsque l'on double la taille des données :

1. Pour un coût logarithmique $C(n) = a \log n$, on a :

$$C(2n) = a \log(2n) = a \log n + a \log 2 = C(n) + a \log 2,$$

donc *le coût augmente d'une constante lorsque l'on double la taille des données*.

2. Pour un coût linéaire $C(n) = an$, on a :

$$C(2n) = a(2n) = 2an = 2C(n),$$

donc *le coût double lorsque l'on double la taille des données*.

3. Pour un coût logarithmique $C(n) = n^2$, on a :

$$C(2n) = (2n)^2 = 4n^2 = 4C(n),$$

donc *le coût quadruple lorsque l'on double la taille des données*.

Pour un algorithme tel que $C(n) = an \log n$, on a :

$$C(2n) = 2an \log(2n) = 2an \log n + 2an \log 2 = 2C(n) \left(1 + \frac{\log 2}{\log n}\right) \sim 2C(n).$$

La manière dont le coût augmente lorsque l'on double la taille des données est donc *asymptotiquement* la même que pour un algorithme de coût linéaire. Pour de grandes données, cela revient donc pratiquement au même.

L'algorithme de fusion

Cet algorithme est destiné à servir de procédure à l'intérieur d'un autre algorithme : nous utiliserons donc des notations un peu plus générales pour les indices et les noms des tableaux. On suppose donnés un tableau trié de ℓ éléments $T[a+1], \ldots, T[a+\ell]$ et un tableau trié de m éléments $U[b+1], \ldots, U[b+m]$. L'algorithme les fusionne et remplit un tableau de $n := \ell + m$ éléments $V[c+1], \ldots, V[c+n]$, qui sera trié (nous notons `tab1` et `tab2` les tableaux à fusionner, `tab` celui qui reçoit le résultat de la fusion) :

```
 i := a + 1;
 j := b + 1;
 k := c + 1;
 tant que (i <= a + l et j <= b + m) faire
   si tab1[i] < tab2[j] alors
      res[k] := tab1[i];
      i := i+1;
      k := k+1;
   sinon
      res[k] := tab2[j];
      j := j+1;
      k := k+1;
   fin_si;
 fin tant que;
 tant que (i <= a + l) faire
    res[k] := tab1[i];
    i := i+1;
    k := k+1;
 fin tant que;
 tant que (j <= b + m) faire
    res[k] := tab2[j];
    j := j+1;
    k := k+1;
 fin tant que;
```

(Sur les deux dernières boucles, l'une est vide car la condition de sortie de la première boucle est que l'un des deux tableaux T,U ait été entièrement parcouru.) Le nombre total d'affectations de $V[k]$ est exactement n, le nombre de comparaisons est majoré par n dans tous les cas.

L'algorithme de tri

L'algorithme de fusion ci-dessus ne peut être effectué "sur place" : autrement dit, si l'on fusionne par exemple $T[0],T[1]$ avec $T[2],T[3]$, le résultat ne peut être directement écrit dans $T[0],T[1],T[2],T[3]$. La manière la plus évidente de l'exploiter est d'écrire les résultats dans un tableau U, puis de recopier U dans T, et ceci à chaque étape. Voici le *schéma* de l'algorithme complet (où l'on note `tab` et `aux` les deux tableaux utilisés) :

```
pour i dans 0..k-1 faire
   pour j dans 0..2^(k-i-1) faire
      fusionner tab[2j.2^i..(2j+1).2^i - 1] et tab[(2j+1).2^i..(2j+2).2^i - 1]
         dans aux[2j.2^i..(2j+2).2^i - 1];
   fin pour;
   recopier aux dans tab;
fin pour;
```

Quelques remarques finales :

1. Pour la première fois, nous utilisons une boucle "pour" au lieu d'une boucle "tant que" : c'est bien adapté à un parcours de tableau.

2. Nous avons noté `T[p..q]` le sous-tableau de T formé des éléments d'indices allant de p à q.
3. On peut améliorer l'algorithme en évitant les recopies ; il suffit pour cela de "basculer" à chaque étape le tableau source et le tableau but : la première phase de fusion se fait de T dans U, la deuxième de U dans T, etc, en alternant.

TP 4.2.6 Ecrire et programmer l'algorithme avec alternance des tableaux source et but.

4.3 Suppléments facultatifs

L'arbre des choix et la minoration du coût d'un tri par comparaisons.

Principe (non détaillé) d'un algorithme dichotomique et son coût.

Annexe A

Rappel de la proposition

Jacques Sauloy. Université Paul Sabatier. UFR MIG. Hiver 2011

Proposition [1] de contenu pour l'option Math/Info

- C'est conçu avec l'idée (approximative) de douze séances de cours-TD de 1h30 et autant de TP, mais commençant un peu plus tard.
- La bibliographie indiquée ci-dessous est pour nous enseignants, sauf [4] (modules II.1 et II.8, ainsi qu'une partie du module I.1), qui concerne les étudiants.

A propos des calculs de coût. Les calculs de coût sont toujours faits dans le cas le pire et le cas le meilleur ; et en moyenne quand c'est possible. Le but pédagogique n'est pas de montrer que tel algorithme est plus performant, mais que l'on peut prévoir le comportement d'un algorithme au prix d'un raisonnement mathématique. Il faudra insérer une discussion des classes (logarithmique, linéaire, quasi-linéaire, polynomiale, exponentielle) et peut-être quelques expériences en TP pour observer les comportements correspondants.

A.1 Arithmétique élémentaire (quatre semaines)

A.1.1 Ecriture et calcul en base b

Division euclidienne (rappel).
Ecriture en base b. Algorithme de conversion. Taille d'un entier.
Addition et multiplication en base b. Coût.

A.1.2 Exponentiation

Exponentiation : normale, dichotomique, optimale, aberrante. Correction. Coût simple, coût affiné.

1. Version révisée après la réunion du 31/01/2011.

A.1.3 Algorithme d'Euclide

L'algorithme d'Euclide de calcul du pgcd. Correction. Coût.
L'algorithme d'Euclide étendu (avec calcul des coefficients de Bézout). Coût.

Suppléments facultatifs envisageables. Résolution de l'équation $ax+by=c$.
Nombres premiers. Le crible d'Eratosthène.

A.2 Arithmétique modulaire et applications (quatre semaines)

A.2.1 Congruences

Calcul modulo m : addition, multiplication.

A.2.2 Générateurs pseudo-aléatoires

Générateur pseudo-aléatoire "linéaire congruentiel".

A.2.3 Cryptographie

Puissances. Logarithme discret. RSA.

Suppléments facultatifs envisageables. Recherche de la période (méthode de Brent).
Résolution de l'équation $ax \equiv b \pmod{m}$. Résolution simultanée (lemme chinois).
Tests de primalité, factorisation (méthodes élémentaires).

A.3 Ensembles finis (deux semaines)

Bijection entre $\mathcal{P}(A)$ et $\{0,1\}^A$.
Calcul dans $(\mathbf{Z}/2\mathbf{Z})^n$ de : complémentaire, intersection, réunion, différence symétrique, inclusion.
Codage du produit de deux ensembles.

Suppléments facultatifs envisageables.

A.4 Recherche et tri (deux semaines)

A.4.1 Algorithmes de recherche (vuis en cours d'info)

Modèle probabiliste utilisé.
Recherche d'un élément dans un tableau quelconque, dans un tableau trié. Correction. Coût.
Recherche du maximum. Correction. Coût.
Recherche d'un duplicata dans un tableau quelconque, dans un tableau trié. Coût.

A.4.2 Algorithmes de tri

Tri de trois éléments. Coût.
Tri par sélection (vu en cours d'info). Correction. Coût.
Tribulle. Correction. Coût. Lien avec les inversions d'une permutation.

Suppléments facultatifs envisageables. L'arbre des choix et la minoration du coût d'un tri par comparaisons.
Principe (non détaillé) d'un algorithme dichotomique et son coût.

Bibliographie

[1] **T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein.** *Introduction à l'algorithmique.* Dunod, 1994.

[2] **M. Demazure.** *Cours d'algèbre. Primalité, divisibilité, codes.* Cassini, 1997.

[3] **D. E. Knuth.** *The Art of Computer programming, vol. II, Seminumerical Algorithms.* Addison-Wesley, 1081.

[4] **J.-P. Ramis et A. Warusfel.** *Mathématiques. Tout-en-un pour la licence, niveau L1.* Dunod, 2006.

[5] **J.-P. Ramis et A. Warusfel.** *Mathématiques. Tout-en-un pour la licence, niveau L2.* Dunod, 2007.